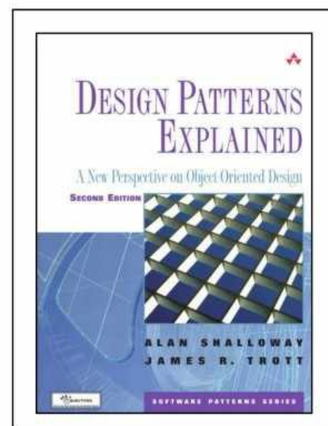
PEARSON



软件开发方法学

精选系列



Design Patterns Explained:
A New Perspective on Object-Oriented
Design, Second Edition

[美] Alan Shalloway James R. Trott 著

徐言声 译

设计模式解析

（第2版·修订版）

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

目　录

软件开发方法学精选系列

Design Patterns Explained:A New Perspective on Object-Oriented Design,Second Edition

设计模式解析（第2版·修订版）

［美］Alan Shalloway　James R.Trott　著

徐言声　译

人民邮电出版社

北京

图书在版编目（**CIP**）数据

设计模式解析/（美）沙洛维（Shallowav,A.），（美）特罗特（Trott,J.R.）著；徐言声译.--2版（修订本）.--北京：人民邮电出版社，2016.1

（软件开发方法学精选系列）

书名原文：Design Patterns Explained:A New Perspective on Object-Orientde Design,Second Edition

ISBN　978-7-115-41014-6

Ⅰ.①设…　Ⅱ.①沙…②特…③徐…　Ⅲ.①软件设计

Ⅳ.①TP311.5

中国版本图书馆CIP数据核字（2015）第279417号

内容提要

本书以作者自身学习、使用模式和多年来为软件开发人员（包括面向对象技术老兵和新手）讲授模式的经验为基础撰写而成。首先概述了模式的基础知识，以及面向对象分析和设计在当代软件开发中的重要性，随后使用易懂的示例代码阐明了12个最常用的模式，包括它们的基础概念、优点、权衡取舍、实现技术以及需要避免的缺陷，使读者能够理解模式背后的基本原则和动机，理解为什么它们会这样运作。

本书适合软件开发专业人士，以及计算机专业、软件工程专业的高校师生阅读，也可作为面向对象分析与设计课程的参考书。

◆著　［美］Alan Shalloway　James R.Trott

译　徐言声

责任编辑　杨海玲

责任印制　张佳莹　焦志炜

◆人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

北京艺辉印刷有限公司印刷
◆开本：800×1000　1/16
印张：19.5
字数：335千字　　2016年1月第2版
印数：3001-5500册　　2016年1月北京第1次印刷
著作权合同登记号　图字：01-2012-7096号
定价：55.00元
读者服务热线：**(010)81055410**　印装质量热线：**(010)81055316**
反盗版热线：**(010)81055315**
广告经营许可证：京崇工商广字第**0021**号

版权声明

其他

献给Leigh、Bryan、Lisa、Michael和Steven，为了他们的爱、支持、鼓励和牺牲精神。

——Alan Shalloway

献给Jill、Erika、Lorien、Mikaela和Geneva，你们是我生命花园中的玫瑰。

唯上天得荣耀。

——James R.Trott

前言

如果我已经有了第1版，还需要买第2版吗？

回答当然是“需要”！原因如下。

自完成第1版的写作之后，我们对设计模式又有了大量更深入的理解，包括以下一些方面。

如何使用共性和可变性分析来设计应用程序的架构。

设计模式与极限编程（eXtreme programming，XP）和敏捷开发的关系，以及设计模式如何有助于二者的实施。

为什么测试是高质量编程的一个优先原则。

为什么使用工厂（factory）实例化和管理对象至关重要。

对帮助学生理解如何用模式思考而言，哪些模式是必不可少的。

本书探讨了所有这些主题。我们进一步深化和澄清了前一版阐述过的主题，并增加了一些非常有用的新内容，包括：

第15章，共性和可变性分析；

第20章，来自设计模式的教益：各种工厂模式；

第22章，Object Pool模式（《设计模式》一书中没有讨论的模式）；

第24章，工厂模式的总结。

我们改变了一些模式的阐述顺序，据参加该课程的学生反映，这样的顺序更有助于掌握模式背后的思想。

所有章节的内容都进行了少量修改，综合了过去三年来从许多读者那里收到的各种反馈意见。而且，为了帮助学生学习，我们为每一章都编写了复习题（答案在本书配套网站可以找到）。

我们可以非常坦率地说，本书无疑是少数值得拥有的第2版，即使读者已经购买了第1版。

非常乐于倾听您宝贵的建议。

——Alan和Jim

设计模式和面向对象程序设计曾经做出过这样的承诺：要简化软件设计人员和开发人员的工作。技术媒体甚至大众媒体每天都在传播相关的术语。然而，要学习这两种技术，熟练掌握它们并且知其所以然，可能非常困难。

你使用某种面向对象或者基于对象的语言可能已经多年，可你是否知道，对象真正的威力并不在于继承，而是来自封装行为。你可能对设计模式很好奇，而且感觉相关的著作都有些太过深奥和夸张。倘若如此，本书正适合你。

本书是以作者多年来为许多软件开发人员（包括面向对象技术老兵和新手）讲授模式的经验为基础撰写而成的。我们相信，而且我们的经验也已经证明，如果能够理解模式背后的基本原则和动机，理解它们为什么会这样运作，那么你的学习曲线将不可思议地缩短。而且在我们对设计模式的讨论中，可以懂得真正的面向对象思维定式，这正是登堂入室的必由之路。

通过阅读本书，读者能够完整地理解12个核心设计模式和1个分析模式，也将了解到设计模式并不是孤立存在的，多个设计模式协同工作才能帮助我们创建更加“健壮”的应用程序。你还可以获得阅读其他设计模式文献所需的足够基础知识，如果愿意，可能还能够自己发现新的模式呢。最重要的是，你将为创建灵活、完善而且更易维护的软件做好准备。

虽然这里所讲授的12个模式并没有涵盖所有应该学会的模式，但是理解了这12个模式，就能够举一反三，更加容易地自学其他模式。我们没有讨论入门所需之外的更多模式，而是讲述了更加有用的与模式相关的若干问题。

从面向对象到模式再到真正的面向对象

从很多方面来看，本书实际上是在复述我自己学习设计模式的经历。我是在学习模式本身之后，再学习模式背后的思想。然后，又将这种理解扩展到分析和测试领域，也扩展到学习模式与敏捷编程方法的关系中。本书第2版中包含了许多第1版出版后我的一些领悟。在学习设计模式之前，我自认为已经是一个很不错的面向对象分析和设计专家了，我曾经设计和实现了几个针对许多不同行业的非常出色的项目。我使用C++并且正在学习Java，代码中的对象可以说是中规中矩[1]、封装严密，而且还能为继承层次结构设计出优秀的数据抽象。我想自己应该已经得面向对象之道了。

现在回想起来，我发现自己那时虽然总是遵循大多数专家的建议行事，但是并没有真正理解面向对象设计的全部威力。直到开始学习设计模式，我的面向对象设计能力才得以拓展和加强。即使还没有直接使用模式，理解设计模式也已经使我成为更加出色的设计人员。

我开始学习设计模式是在1996年。那时我还是美国西北部一家大型航天公司的一名C++和面向对象设计讲师。有几个人要求我领导一个设计模式学习小组，正是在那里我遇到了本书的另一位作者Jim Trott。学习小组中发生的几件事情很有意思。一开始，我就对设计模式着了迷。我很喜欢能够将自己的设计与其他经验更多的人的设计进行比较。然后，我就发现自己并没有发挥“按接口设计”（designing to interface）的全部威力，而且并不总是关注是否存在“一个对象在还不知道另一个对象的类型时就使用这个对象”的情况。我还注意到，刚刚从事面向对象设计的人（一般总是认为这时就学习设计模式过早）从学习小组所得的

获益，居然同专家差不多。设计模式展示了优秀的面向对象设计实例，而且阐明了基本的面向对象设计原则，这些有助于初学者更快设计出成熟的方案。到整个学习结束时，我已经完全相信：设计模式是面向对象设计发明以来软件设计领域出现的最伟大的东西。

可是，在我审视当时自己的工作时，却看到代码中并没有使用任何设计模式。或者说，至少还没有有意识地使用任何一个模式。后来，随着对模式学习的深入，我发现自己开始在代码中使用许多设计模式了，不再仅仅是一个好的编程匠。当然，现在我对模式的理解更加深入，使用起它们来也更加得心应手了。

我当时只是觉得自己可能对设计模式还了解得不够，应该学习更多。那时我只了解其中6个模式。然后，我突然顿悟。当时我是一个项目的面向对象设计顾问，而且应要求为项目做高层设计。这个项目的负责人极为聪明，但在面向对象设计方面却是一个新手。

问题本身并不怎么难，但是对代码维护性的要求很高。我花两分钟查看了一下问题，然后就照本宣科地按照通常使用的数据抽象方法提出了一个设计。很不幸，我自己也清楚这不会是什么好的设计。只用数据抽象使我无功而返，必须另寻良策。

两个小时之后，我用尽了自己知道的所有设计技术，已经黔驴技穷，情况却毫无好转的迹象。我的设计没有本质上的变化。最让人感到灰心的是，我知道肯定有更好的设计方案，只是我想不出来。更具讽刺意味的是，我还知道这个问题里藏着4个设计模式，可是我不知道怎么使用。于是，我——一个自封的面向对象设计专家，被一个简单问题生生噎住了！

我感到失望之极，只好休息一下，出去走走，清醒清醒头脑。我告诉自己，至少10分钟之内不要再想这个问题了。可是，才过30秒钟，我就忍不住又思考起来！这次我突生灵感，刹那间，自己的设计模式观改变了：不能将模式作为一个单独的东西使用，应该把它们结合起来。

模式应该相互配合，共同解决问题。

这话以前我也听说过，但是当时并没有真正理解。因为软件中的模式最初以设计模式为名引入，我想当然地认为它们主要都是关于设计的，并一直囿于这样的想法而碌碌无为。我曾认为，在设计领域，模式就是类之间合乎规则的关系。后来读到Christopher Alexander的奇书The Timeless Way of Building（牛津大学出版社，1979）[2]，我才知道所有层次——分析、设计和实现都存在模式。Alexander曾阐述了使用模式有助于理解问题域（甚至有助于描述它），而不仅仅是用来在理解问题域之后完成设计。

我错就错在试图在问题域中创建类，然后再将它们结合起来形成一个系统，这正是Alexander所说的非常糟糕的做法。我从没有自问过这些类是否正确，因为它们看上去很好，显而易见，类从一开始分析就马上浮现于我的脑海，而且它们正是按教科书一直教的那样应该在系统描述中寻找的那些“名词”。但是试图将它们组合起来时却遇到了重重困难。

当我再回到办公室，在设计模式和Alexander方法的指导下重新创建类时，仅仅几分钟时间，一个大为改观的解决方案就展现出来。这真是一个好设计，我们最后根据它实现了产品。我真的很兴奋——兴奋于自己设计了如此优秀的方案，也兴奋于设计模式的威力。从那时起，我开始将设计模式融入开发工作和教学中。

我发现，不熟悉面向对象设计的程序员也可以学习设计模式，而且，通过学习设计模式，他们能够掌握基本的面向对象设计技术。对我而言就是如此，对我教授的学生而言亦然。

想象一下我是多么惊讶吧！我曾经读过的设计模式图书，曾经与之交谈过的设计模式专家都说，在开始设计模式研究之前，需要扎扎实实地打好面向对象设计基础。可是，我自己的亲身经验却表明，在学习面向对象设计的同时学习设计模式的学生，比仅学习面向对象设计的学生进步更快，他们掌握设计模式的速度甚至与有经验的面向对象老手一

样。

我开始使用设计模式作为自己教学的基础，并且将自己的课程称为“面向模式的设计：设计模式从分析到实现”。

我希望我的学生能理解这些模式，继而可以发现使用一种探索式的方法是有助于促进这种理解的最佳方式。例如，我发现，先提出问题，然后通过大多数模式中都适用的一些指导性原则和策略，帮助学生尝试为此问题设计出解决方案，这样阐述Bridge（桥接）模式更好。在实际探索中，学生找到了解决之道——其实就是Bridge模式，并牢牢地记在心中。

设计模式与敏捷方法/极限编程

设计模式之下所隐藏的指导性原则和策略现在对我而言已经非常清楚了。《设计模式》一书中肯定提到了这些内容，但是讲得太简洁，以至于我第一次阅读时完全没有体会到其价值。我相信《设计模式》一书实际上是以Smalltalk社区为目标读者而写的[3]，这些原则在Smalltalk社区可以说是根深蒂固，所以不需要太多背景。但是因为自己对面向对象范型的理解很有限，我理解这些原则花了很长时间。直到我将《设计模式》四位作者的工作与Alexander的工作、Jim Coplien关于共性和可变性分析的工作、Martin Fowler关于方法学和分析模式的工作结合起来之后，这些原则对我而言才变得足够清晰，我甚至能够向其他人讲解。这对我自己的教学生涯也很有帮助——我再也不能像自己工作时那样容易地想当然，而又能侥幸无事了。

自本书第1版出版以来，我一直在进行大量的敏捷开发实践，具有了较多的极限编程实践、测试驱动开发（TDD）和Scrum的经验。刚开始，在结合设计模式和极限编程、测试驱动开发时还是经历了一段困难时期。但是，我很快认识到它们都非常重要，而且植根于一些相同的原

则（虽然设计方法并不相同）。事实上，在敏捷软件开发训练课上，我明确说明，如果正确使用设计模式将为引入敏捷开发打下很好的基础。

贯穿本书始终，我讨论了许多设计模式与敏捷管理和编程实践的关联。如果读者对极限编程、测试驱动开发或者Scrum不熟悉，可以不用太在意这些论述。但是，如果真的如此，我建议你下一步就去读一本有关的著作。

我发现无论何种情况下，都可以用这些指导性原则和策略来“推导”出几个设计模式。这里“推导出设计模式”的意思是，如果看到可能用设计模式解决的问题，就可以使用从模式中学到的这些指导性原则和策略，得到以模式表达的解决方案。我明确地告诉学生们，我们并不是用这种方法真正得出设计模式；相反，我只是要说明一种可能的思考过程，最终成为设计模式的那些解决方案的提出者使用的也是这样的过程。

我解释这些数量不多但很强大的原则和策略的能力与日俱增。随之而来的，是我发现自己解释《设计模式》一书中的模式时，它们更加有用了。事实上，我在设计模式课上用这些原则和策略能够阐述几乎所有的模式。

我还发现，无论是否使用设计模式，自己在设计中都在使用这些原则。我对此并不感到惊讶。如果使用这些原则和策略能够得到后来才发现等效于设计模式的设计，那么就说明这些原则和策略已经给了我一种自己做出优秀设计的方法（因为根据定义，模式代表着优秀设计）。有了这些技术难道还会只是因为不知道对应模式（可能已知也可能还没有发现）的名字而得出较差的设计吗？

这些认识帮助我很好地磨砺了自己的培训过程（和现在的写作）。我现在的教学工作已经有了好几个层次。教授面向对象分析和设计基础时，我教设计模式，并用它们作为优秀面向对象分析和设计的例子。此外，通过设计模式来教授面向对象概念，学生们还能更好地理解面向对

象原则。而且教授指导性原则和策略，使学生们可以自己创建出质量能够与模式媲美的设计。

在此讲述这些，是因为本书正是沿袭了我所教授课程的模式，几乎所有的材料就是我们目前的课程之一——“设计模式、测试驱动开发或者敏捷开发最佳实践”[4]的内容。

通过阅读本书，你将学习到这些模式。但尤其重要的是，你将学到模式为何有效和如何协同工作，以及模式背后的原则和策略，这将有助于充分利用你自身的经验。当本书提出一个问题时，如果你能够联想到一个曾经碰到的类似问题，将极其有益。本书并没有讲述什么新知识或者如何应用新模式，而是提供了一种考虑面向对象软件开发的新视角。我希望你的自身经验能够与设计模式的原则结合，成为学习过程中强大的助力。

Alan Shalloway
2000年12月第1版
2004年10月第2版

从人工智能到模式再到真正的面向对象

我的设计模式历程与Alan的设计模式历程可以说是殊途同归，具有下述同样的结论。

基于模式的分析能够使我们成为更高效也更有效的分析人员，因为它使我们能够更加抽象地处理模型，因为它代表了许多其他分析人员的集体经验。

模式能够帮助人们学习面向对象的原理，并有助于解释我们处理对象的方式。

我的职业生涯开始于人工智能领域，工作是创建基于规则的专家系统。这其中涉及倾听专家们的讲述，为其决策过程建立模型，然后将这些模型编码成基于知识的系统中的规则。在构建这些系统时，我发现了一些反复出现的主题。对于一些相同类型的问题，专家们总是用类似的

方法去解决。例如，诊断设备问题的专家往往首先寻找简单、快速的解决方案，然后再系统化一些，通过系统分析将问题分解为多个子问题。在系统诊断时，他们也往往在进行其他形式的测试之前，先尝试成本低或者能排除较大范围问题的测试。其实无论诊断的是计算机中的还是某个油田设备的问题，这种方法都适用。

要换在今天，我会把这些反复出现的主题称为“模式”。设计新的专家系统时，我很自然地也开始寻找这些反复出现的主题了。对于模式思想，我当时是非常开放而且心存好感的，虽然还不知道它们是什么。

后来到了1994年，我发现欧洲的研究者们已经整理了这些专家行为的模式，放入名为“知识分析和设计支持”（简称KADS）的软件包中。Karen Garder博士——一位才华横溢的分析师、建模专家、顾问，一个天才，开始在美国将 KADS 应用于她的工作中。她扩展了欧洲研究者的工作，将 KADS 应用于面向对象系统。她使我的眼界大开，看到了软件界正在形成的一个全新领域——基于模式的分析和设计，这很大程度上都源自 Christopher Alexander 的工作。Karen Garder的书Cognitive Patterns（剑桥大学出版社，1998年）叙述了这些思想。

突然之间我有了一个为专家行为建模的框架，能够不至于过早地陷入复杂和异常情况。借助这种框架完成接下来的三个项目时，我花费的时间缩短了，重复工作减少了，而最终用户的满意度却提高了，原因如下。

我能更快地设计模型，因为模式已经事先告知会出现什么情况。它们能够告诉我基本的对象是什么，什么应该特别注意。

我与专家沟通的成效大增，因为对于处理细节和异常情况我们有了结构化更强的方法。

通过模式，能够针对系统设计更好的最终用户培训方案，因为模式已经预先告知系统最重要的特性。

最后一点意义重大。模式之所以能够帮助最终用户理解系统，是因

为它们提供了系统的来龙去脉，说明了为什么我们会这样设计。我们可以用模式描述系统的指导性原则和策略。我们可以用模式开发最佳范例，从而帮助最终用户理解系统。

我被这一切吸引住了。

因此，当我就职的公司组织了一个设计模式学习小组时，我当然积极参与。于是我遇到了本书的另一位作者Alan　Shalloway，他作为一位面向对象设计师和顾问也殊途同归地在工作中体会到了类似的境界。于是也就有了本书的诞生。

自完成第1版以来，我进一步领悟到这种分析方法能够多么深刻地增进我们的理解。我参与了许多不同类型的项目，其中许多甚至与软件开发无关。我看到许多一起协作、相互交换知识、交换思想、生活在不同地方的人所组成的各种系统。模式和面向对象的原则依然有助于我。和计算机系统中一样，减少工作系统之间的依赖性也能获得很好的效果。

我衷心希望本书中讲述的原则能够对各位读者成为更有效而且更高效的分析人员有所帮助。

James R.Trott

2000年12月第1版

2004年12月第2版

本书约定

在本书写作过程中，我们选用了一些特定的风格和版式约定。读者可能对其中一些不太习惯。因此我们对如此选择的原因作以下说明。

第一人称

本书是两位作者的合作作品。为了找到阐述这些概念的最佳方式，我们经常争论并且不断做出改进。Alan 在他的课程中曾使用这些方式试讲，然后我们又共同进行了改进。本书主体部分中我们选择使用第一人称单数，因为这能够如我们希望的那样，以更生动和自然的方式娓娓道

来。

便于浏览

我们试图使本书易于浏览，读者能够在不全部阅读正文的情况下获得要点，也可以迅速找到需要的信息。我们大量使用了表格和带有项目符号的列表并在页边加上了总结相应段落的文字。讨论每个模式时，提供了模式关键特性总结表。我们相信这些措施能够使本书的可读性大大提高。

示例代码

本书讲述的是分析和设计，而不是具体的代码实现，其目的是帮助读者在面向对象开发中积累的真知灼见和最佳实践经验的基础上，思考如何做出优秀的设计，就像用设计模式所表示的那样。所有程序员都会遇到的挑战之一就是：不能过早开始进行实现，需要三思而后行。既然如此，本书有意地尽量避免过多讨论实现。我们的示例代码可能看上去有些分量不足而且不够完整。比如说，代码都没有提供错误检查。这是因为使用它们只是为了说明概念。然而，本书的配套网站http://www.netobjectives.com/dpexplained中存放了更完整的示例代码①，书中的代码是从中摘出来的。C++语言和C#语言的示例代码也可以在这个网站中找到。

策略和原则

本书是一本介绍性读物，有助于你快速学习设计模式，并从中理解启发了设计模式的原则和策略。阅读本书之后，可以继续阅读更学术化的模式图书或参考书。本书最后一章将列出许多可能对你有用的参考书。

展现广度，形成认识

本书努力使读者能够对设计模式有所认识，书中将展现模式领域的广度，但不会很深入地讲述任何一个模式（参见上一点）。

如果带一个人去美国旅游两个星期，你会带他去哪里呢？也许是几

个主要景点，可以帮助他了解一些建筑、风土人情、城市及城市之间的广袤原野的风光、高速公路和咖啡厅，但是不可能向他展示一切。为了丰富他的知识，可以选择性地给他播放一些其他景点和城市的宣传片，使他有所认识。以后他就可以自己计划未来的旅游行程。我们试图让你对设计模式有一个基本认识，因而与此类似，本书也将带你参观设计模式中的主要景点，然后使你对其他方面也有所认识，这样读者就可以自己计划进一步的设计模式学习旅程了。

C#开发人员如何阅读**Java**代码

本书中的所有代码都是用Java语言编写的。如果你没有使用Java的经验，但是能够阅读C#代码，需要了解以下几点：

Java使用 extends和 implements分别表示扩展另一个类或者实现一个接口的类，而不是像在C#中那样两种情况都用冒号（:）。

因此，在Java程序中，我们会看到：

public class NewClass extends BaseClass

或者

public class NewClass implements AnInterface

而在C#程序中，看到的则是：

public class NewClass : BaseClass

或者

public class NewClass : AnInterface

Java中的所有方法都是虚拟的，因此不用指定方法是new还是overridden。Java中没有这样的关键字，所有子类方法都会改写它们从基类继承的方法。虽然还有其他区别，但是我们的代码示例中不会出现。

C++开发人员如何阅读**Java**代码

C++开发人员阅读Java要困难一些，但是也不算太难。最明显的区别是Java没有头文件。但是阅读组合了头文件和代码的Java式文件，其实非常直观，无需解释。除了上述与C#的区别之外，Java是不在栈中存储对象的。Java在堆存储区中存储对象，栈中只存储保存着对象引用（指针）的变量。所有对象都必须用new关键字创建。

因此，在Java程序中，我们会看到：

```
MyClass anObject= new MyClass();
anObject.someMethod();
```

而在C++中，看到的则是：

```
MyClass *anObject= new MyClass();
anObject->someMethod();
```

因此如果在每个引用对象的变量名声明中添加一个星号（*），然后将句点（.）转换为连字符加右尖括号（->），Java代码看上去就像C++代码了。

反馈

设计模式仍然在发展当中，它们其实就是发现最佳实践以及面向对象基本原则的专业人员之间的行话。

因此，我们非常重视你对本书的反馈意见。

我们什么地方做得比较好，什么地方做得还不够？

有没有需要改正的错误?

有没有什么地方写得不够清晰?

请访问Net Objectives公司为本书英文版设立的配套网站，网址是http://www.netobjectives.com/dpexplained。在这个网站上，能够找到我们的最新研究成果，以及与本书和一般性软件开发问题相关的讨论组。请在讨论组中发表改正意见、评价、真知灼见和心得体会。

第2版的新内容

第2版相对第1版而言有许多变化和改进。它反映了我们过去几年使

用和教授设计模式中的收获，也加入了从读者那里收集到的无私和非常有价值的反馈意见。

主要的变化有下面几个方面。

重新组织了章节顺序（例如，将对Strategy模式的叙述提前）。

扩展了对共性和可变性分析（CVA）的讨论。

将极限编程和设计模式结合起来。

所有示例现在都是完整的可执行代码，而不再是示意性的或者片段代码了。配套网站还有C#和C++语言示例代码。

增加了对为什么将工厂用作实例化器/管理器的解释，这部分内容极为有用。

新增一个《设计模式》书中没有讲到的模式：Object Pool。

讨论了模式的缺陷，包括关于“将模式作为辅助思考指导”的警告。模式并不是真理！

在语法和风格方面也进行了大量小的订正。

致谢

几乎每本书前言的最后都会有致谢，感谢帮助该书出版的人。直到自己写书，我们才如此真切地体会到其中缘由。出一本书的确是集体劳动的结晶。我们要感谢的人，可以列出一个长长的清单。

以下诸位对我们的帮助尤其重要。

Addison-Wesley公司的Debbie Lafferty，她永不疲倦对我们进行鼓励和督促。

我们的同事Scott Bain，他耐心地审阅了本书，并提供了许多真知灼见。他和Alan在Net Objectives公司的合作对于本书第2版中新增的许多内部都很有启发性。

我们的审稿团队：James Huddleston、Steve Metsker和Clifton Nock。

特别要提到Leigh和Jill——她们是极富耐心的妻子，容忍而且鼓励

我们完成本书，实现梦想。

审阅本书第1版和第2版多个草稿版本的人很多，他们提供了许多极好的评论。我们尤其要提到 Brian Henderson、Bruce Trask、Greg Frank和Sudharsan Varadharajan，他们无私而且耐心地让我们分享了他们自己的所知所想。

Alan要特别感谢——

我以前的几个学生，他们对我的影响之大，可能他们自己都永远不会知道。在我对是否在课程中引入新想法犹豫再三，感觉似乎应该墨守成规的时候，是他们在我刚开始讲述课程时对新概念的热情，鼓励我在课程中越来越多地加入了自己的想法。感谢 Lance Young、Peter Shirley、John Terrell和Karen Allen。他们的行动对我来说是一种不断的提醒，说明鼓励的作用是多么深远。

John Vlissides，感谢他富于思想的意见和启发性的询问。

第2版还要加上对另一位Net Objectives公司的同事Dan Rawsthorne博士的感谢。他从事敏捷开发的方法对我影响很大。对另一位同事Jeff McKenna的支持和肯定我也感激不尽。我还要感谢Kim Aue，我们 Net Objectives公司的“大总管”，她在各方面的支持对我帮助极大。

我要特别感谢Martin Fowler、Ken Schwaber、Ward Cunningham、Robert Martin、Ron Jeffries、Kent Beck和Bruce Eckel就本书相关问题与我的交谈（有时候是通过电子邮件），当然，这并不是意味着本书内容他们都同意或者认可。

Jim要特别感谢——

Karen Gardner博士，人类思维模式顾问和良师。

Marel Norwood博士和Arthur Murphy，我在KADS和基于模式分析方面的最初合作者。

Brad VanBeek，他为我提供了在本学科中发展的空间。

Alex Sidey，他指导我掌握技术写作的规律和诀窍。

Sharon和Bob Foote博士，现在任教于西点军校，使我具备了对人永不知足的好奇心和持久的兴趣。他们的爱和鼓励已经成为模式永存我心，无论是作为一个人、一位父亲和丈夫，还是作为一位分析师。

千禧救助与开发服务组织（www.mrds.org）的Bill Koops和Lindy Backues，他们帮助我看到，基于模式的方法甚至能够用于救助贫穷和边缘化的人。他们真是好伙伴，好导师。

---

[1].原文为well-formed，这个术语许多文献译为“格式/形式良好的”、“良构”等等，但根据C++标准1.4.1的定义：“按照语法规则、可诊断语义规则和定义一次规则构造的C++程序”，似乎按照数学界的习惯译为“合式的”或“合乎规则的”更好。此处因为并不强调技术语义，所以随文就译了。——译者注

[2].中文版《建筑的永恒之道》，由知识产权出版社出版。——译者注

[3].此说法不确切，设计模式的起源与量子力学殊途同归的群英会非常相似，得多人之力，而众多先驱中只有 Kent Beck、Ward Cunningham和Ralph Johnson等是纯Smalltalk社区背景的，Eric Gamma、Jim Coplien、John Vlissides等则来自C++社区。事实上，《设计模式》一书本身是用C++作为描述语言的。——译者注

[4].参见本书英文版配套网站http://www.netobjectives.com/dpexplained，还有更多有关课程的信息。

第一部分 面向对象软件开发简介

概览

本部分内容

本部分将介绍一种以模式（众多设计人员和用户多年来所获得的领悟和最佳实践经验）为基础的面向对象软件开发方法，以及支持此方法的建模语言（UML）。

我不会再采用20世纪80年代的方式，只是告诉开发人员“在需求表述中寻找名词，并将它们转化为对象”。在这种方式中，将“封装”定义为“数据隐藏”，将“对象”定义为“含有数据和用来访问、操作数据的行为的一些东西”。这在当时和现在都是一种局限性很大的观点，其局限在于只是关注“如何实现对象”，这是很不完整的。

本部分将对这些概念的定义进行扩展，并以此为基础讨论另一种面向对象范式。这些扩展的定义都是源自设计模式研究的一些策略和原则的结果。这反映了一种更完整的面向对象观。

章 讨论的主题

1 面向对象范型

介绍对对象的最新理解。

2 UML

统一建模语言（UML）为我们提供了工具，能够以一种图形化的、更易理解的方式描述面向对象设计。

第1章 面向对象范型

1.1 概览

本章内容

本章将通过与大家都熟悉的范型——标准结构化程序设计比较异同的方式，来介绍面向对象范型。

当年，面向对象范型正是为了应对使用标准结构化程序设计遇到的诸多挑战才应运而生的。弄清楚这些挑战，我们才能够更好地看到面向对象程序设计的优点，并更好地理解这一机制。

本章无法使你成为面向对象方法的专家，甚至不会介绍所有基本的面向对象概念。但是，本章将使你为阅读本书其他部分做好准备。本书其他部分将阐释如何像专家所做的那样正确使用面向对象设计方法。

本章中，我们将：

讨论一种常用的分析方法，名为功能分解（functional decomposition）；

探讨需求方面问题和应对需求变更的需要（这可是程序设计中罪恶的渊薮！）；

叙述面向对象范型，并展示其实际应用；

指出一些特殊的对象方法；

提供一个面向对象术语表，列出了本章所用到的重要对象术语。

1.2 面向对象范型之前：功能分解

功能分解是一种处理复杂问题的自然方法

让我们从对一种常用的软件开发方法的考察开始吧。如果给你一个任务，要编写一段代码，访问在数据库中存储的形状描述然后显示出来。按照所需要的步骤来思考，是一种很自然的选择。比如，你可能认为应该按照以下步骤解决这个问题。

1.在数据库中找到形状列表。

2.打开形状列表。

3.按某种规则将列表排序。

4.在显示器上显示各个形状。

还可以选取以上任意一个步骤，进一步分解成实现所必需的若干步。例如，可以将步骤 4 分解。对于列表中所有形状，都可以按照以下步骤进行。

4a.识别形状的类型。

4b.获取形状的位置。

4c.以形状的位置作为参数，调用显示形状的函数。

这种方法就称为“功能分解”，因为分析人员将问题分解成了多个功能步骤（这些步骤就构成了这个问题）。你我都会这样做，因为解决更小的问题，比解决整个问题更简单。这种方法与我写制作意大利肉末番茄烤面条的烹饪过程，或者装配自行车指南所用的方法是一样的。这种方法我们使用得如此驾轻就熟，以至于我们很少对它有所怀疑，或者自问是否还有其他的选择。

这种方法的挑战：能者多责

功能分解方法的一个问题在于，它通常会导致让一个“主”程序负责控制子程序，这是将功能分解为多个子功能的自然结果。但是，主程序所承受的责任太多了：要确保一切正确工作，还要协调各函数并控制它们的先后顺序，因此经常会产生非常复杂的代码。如果让一些子函数负责自己的行为，而且能够告知主函数执行某些任务，并信任它知道如何执行，这种方式比功能分解的方式要容易得多。叱咤疆场的将军和家庭

中成功的父母对这种经验都了然于胸。现在，程序员也学会了，这就是所谓委托（delegation）。

这种方法的难题：应对变化

功能分解方法的另一个问题在于，它在为未来可能出现的变化未雨绸缪方面，在对代码合适地改进方面，都于事无补。变化是无法避免的，经常是因为要为已有的主题增加新的变体。例如，我可能不得不处理新的形状，或者需要显示形状的新方法。如果将实现各步骤的所有逻辑代码都放在一个大函数或大模块中的话，那么这些步骤的任何实质性变化，都必须对这个函数或模块进行修改。

而且变化还会为bug和意料之外的结果创造机会。或者，像我喜欢说的：

许多bug都源于代码修改。

自己去验证这句断言吧。考虑这样的情景：想对代码进行修改，但又害怕这样做，因为你知道修改一个地方的代码可能会破坏其他地方。怎么会出现这种情形呢？代码非要关注所有函数和使用它们的方式吗？函数应该怎样和另一个函数交互呢？函数要关注的细节是否太多了，比如要实现的逻辑、要交互的东西、要使用的数据？和人一样，如果程序试图同时关注过多的东西，一旦有变化出现，就只能坐等bug来到。程序设计可是一种复杂、抽象和动态的活动啊。

而且，无论多么努力工作，无论分析做得多么好，也是永远无法从用户那里获得所有需求的，因为关于未来有太多未知，万物皆变化。不是吗，它们总是在变化之中……

对于阻止变化，我们无计可施。但是我们对变化本身却并非无能为力。

1.3 需求问题

需求总在变化

问问软件开发人员，对于从用户那里获取的需求，他们认为有哪些说法正确。他们经常像下面这样回答。

需求是不完整的。

需求经常是错误的。

需求（和用户）容易让人误解。

需求并不会告诉你全部情况。

只有一种回答你是听不到的："我们的需求不仅完整、清晰、易于理解，而且还说明了我们今后五年需要的所有功能！"

在 30 年编写软件的经历中，关于需求我所学会的主要一点就是，需求总在变化。

我还了解到，大多数开发人员都认为这是一件坏事。但是很少有人能够编写可以很好地处理需求变更的代码。

需求之所以变化，有如下几个简单原因。

用户对自己需求的看法，会因为与开发人员的讨论以及看到软件新的可能性而发生变化。

开发人员对用户问题领域的看法，会在开发使该领域自动化的软件的过程中，因为对它更加熟悉而发生变化。

我可能无法知道什么将会变化，但是我能够猜到在哪里会变化

在刚入行的时候，我有一个师傅总爱说："第二次既然总能编写正确，你第一次也应该能编写正确！"我经常会想起这条忠告。我曾经认为这句话的意思是尝试预期所有可能发生的变化，并相应地构建代码。这当然再好不过，但是通常结果都会令人失望的，因为很少能够预测整个过程所有可能的变化。

最后，我认识到，虽然无法预测会发生什么变化，但是通常可以预

期哪里会发生变化。面向对象的巨大优点之一，就是可以封装这些变化区域，从而更容易地将代码与变化产生的影响隔离开来。

软件开发的环境发生了变化。（5年前谁能想到Web开发能有今日？）这并不意味着我们可以不去收集好的需求。这只是说明我们编写的代码必须要能适应变化，说明我们（可能还有我们的客户）不应该为阻止那些自然而然会发生的事情而庸人自扰。

发生变化了！从容应对

在所有情况下（最简单的除外），需求总会变化的，无论最初的分析做得多好！

与其抱怨需求总是变化，不如改变开发过程，从而更有效地应对变化。

代码可以设计得使需求的变化不至于产生太大影响。代码可以逐步演进，新代码可以影响较少地加入。

1.4 应对变化：使用功能分解

用模块化封装变化

更进一步地来看“显示形状”问题。怎样编写代码才能更容易地应付多变的需求呢？与其编写一个大函数，不如使之更加模块化。

例如，在前面提到的步骤4c“以形状的位置作为参数，调用显示形状的函数”中，可以写一个例1-1所示的模块。

例**1-1** 用模块化封装变化

函数：显示形状

输入：形状类型，形状描述

操作：

switch (形状类型)

case 方形：调用显示方形的函数

case 圆形：调用显示圆形的函数

功能分解方法中模块化的问题

然后，在接到一个需求，要显示新的形状（例如三角形）时，我只需改变这个模块即可。（希望如此！）

但是这种方法仍然存在问题。比如，前面说过，这个模块的输入是形状的类型和描述。然而，在不同的形状存储方式下，对所有形状都适用的一致描述可能存在，也可能不存在。如果形状的描述有时以包含坐标点的数组方式存储，有时以其他方式存储，怎么办呢？这种方法还适用吗？

模块化肯定有助于提供代码的可理解性，而容易理解将使代码更容易维护。但是模块化并不总是有助于代码应对所有可能遇到的变化。

低内聚，紧耦合

这种方法我一直在使用，我发现它主要有两个问题，按术语来说就是低内聚（weak cohesion）和紧耦合（tight coupling）。在Code Complete一书（Microsoft Press, 1993）中，Steve McConnell对内聚性和耦合性有很精彩的描述。他说：

内聚性（cohesion）指的是“例程中操作之间联系的紧密程度”[1]。

我还听说过有人将内聚性称为清晰性（clarity），因为例程（或类）中的多个操作联系越紧密，就越容易理解其含义。说一个类低内聚，指的就是它任务很多而且互不相关，代码经常看上去像是令人疑惑的一大团。最极端的情形下，这些类会与系统中差不多所有东西都纠缠在一起，据说有人称之为“上帝对象”，因为它们好像是万能的（或许是因为只有上帝才能理解它们）。

耦合性（coupling）指的是“两个例程之间联系的紧密程度。耦合性与内聚性是相辅相成的关系。内聚性描述的是一个例程内部组成部分之

间相互联系的紧密程度，而耦合性描述的是一个例程与其他例程之间联系的紧密程度。软件开发的目标应该是创建这样的例程：内部完整（高内聚），而与其他例程之间的联系则是小巧、直接、可见、灵活的（松耦合）。”[2]

修改一个函数甚至是函数所用的数据，都可能对其他函数产生严重破坏

大多数程序员都会有这样的经验：在代码的某个地方修改了一个函数或一个数据，后来却对代码的其他地方造成了意想不到的影响，这种bug称为“不良副作用”。这是因为，虽然我们获得了希望的结果（进行了修改），但是也得到了不需要的结果——bug！更糟糕的是，这些bug经常难以发现，因为我们一开始往往不会注意到那些导致副作用的代码联系（如果能够注意到这些联系，就不会用这种方式修改程序了）。

事实上，这种bug使我有了一个非常惊人的发现：我们实际上并没有花费很多时间改正程序的bug。

我认为，在维护和调试过程中，改正bug只需要花费很少的时间。维护和调试的绝大多数时间都被用于努力弄清代码的运作机理、寻找bug和防止出现不良副作用上了，真正的改正时间却相当短!

因为不良副作用经常是最难发现的bug，所以如果让一个函数处理很多不同的数据，一旦需求发生变化，就更可能出问题。

问题就出在副作用中
只关注函数，就可能引起难以发现的副作用。 维护和调试中所耗费的大多数时间不是花在修改bug上，而是花在寻找bug，弄清如何避免在修改代码时导致不良副作用上了。

功能分解将注意力放在错误的地方

使用功能分解时，需求变更会对软件开发和维护工作产生极大影

响。这时候，主要的精力都放在函数上了，而对一组函数或者数据的修改会影响到其他函数和数据，并依此类推地影响到其他必须修改的函数。就像一个雪球滚下山来，一路裹挟了更多的雪一样，只关注函数，将导致一连串变化，而且难以避免。

1.5 应对需求变更

日常生活中人们如何做事?

为了找出解决需求变更问题的办法，弄清功能分解有没有其他替代方法，我们先来看看日常生活中人们是如何做事的。假设你是要在一个会议[3]上担任讲师，听课的人在课后还要去听其他课，但他们不知道下一堂课的听课地点。你的责任之一，就是确保大家都知道下一堂课去哪里上。

如果按照结构化程序设计的方法，可以按以下的要求做。

1.获得听课人的名单。

2.对于名单上的每个人，做以下工作。

a.找到他或者她要听的下一堂课。

b.找到该课的听课地点。

c.找到从你的教室到下一堂课地点怎么走。

d.告诉这个人怎样去上下一堂课。

为了完成以上工作，你可能需要编写以下内容。

1.获得听课人名单的方法。

2.获得每个人课程表的方法。

3.告诉某个人如何从你的教室到其他任何教室的程序。

4.为听课的每个人服务的一个控制程序，它可以为每个人完成所需的步骤。

你会采用这种方法吗？

我很怀疑是否有人真的会按这样的方法去做。相反，你可能会把从这个教室到其他教室的路线贴出来，然后告诉课堂上的所有人："我已经将下一堂课的地点和其他教室的位置都贴在教室后面了。请根据它找到你们下一堂课的教室。"可以预期每个人都知道自己的下一堂课是什么，而且他们都能从你提供的列表中查到正确的教室，然后按照指示找到它。

这两种方法之间的区别何在呢？

第一种方法——直接给每个人都提供指示，你必须密切关注大量细节，除你之外没有其他人负责。这样你会疯掉的！

第二种方法中，你只给出通用的提示，然后期待每个人会自己弄清怎样完成任务。

责任从你自己转移到每个人……

其中最大的区别就是这种责任的转移。在第一种情况下，你要对一切负责；而在第二种情况下，学生对自己的行为负责。两种情况下，要实现的目的相同，但组织方式差异很大。

这样有什么影响呢？

为了看到这种责任重新安排带来的影响，我们考虑一下在指定了新的需求时情况如何。

假设我被告知，需要给承担助教工作的研究生一些特殊指示。他们可能需要在上下一堂课之前收集本节课的学生评价，并交到会议办公室。在上面的第一种情况下，我将不得不对控制程序进行修改以区分研究生和本科生，然后给研究生特殊指示，从而可能不得不对程序做相当大幅度的修改。

……可以尽量减少变化

而在每个人都各司其责的第二种情况下，我只需要为研究生再编写一个程序，而控制程序仍然只需说"找到你们下一堂课的教室"。每个人只要按此指示相应行事即可。

这代表控制程序的责任发生了明显变化。在第一种情况下，每次需要增加新的一类学生时，控制程序本身都必须作修改，要负责告诉新一类学生如何去做。而在第二种情况下，新一类的学生不会影响控制程序，由学生自己负责弄清如何去做。

为什么会有这种区别呢？

存在不同的原因

这是因为第二种方法有以下三方面不同。

人们对自己的行为负责，而不再由一个中央控制程序负责决定他们的行为。（请注意，为此人们还必须知道自己是什么类型的学生。）

控制程序可以与不同类型的人（研究生和普通学生）交流，好像他们都一样。

控制程序不需要知道学生从此教室到彼教室可能需要采取的任何特殊步骤。

不同的视角

为了完整理解其中的含义，创建一些术语非常重要。在UML Distilled一书（Addison-Wesley，1999）中，Martin Fowler描述了软件开发过程中的三个不同视角（perspective）[4]，如表l-1所示。

表1-1 软件开发过程中的视角

视　　角	描　　述
概念	这种视角“呈现了所研究领域中的各种概念……得出概念模型时应该很少或者不考虑实现它的软件……”。这个视角要回答的问题是：“软件要负责什么？”
规约	“现在我们要考虑的是软件，但我们关注的是软件的接口，而不是实现。”这个视角要回答的问题是：“怎么使用软件？”
实现	这时我们考虑的是代码本身。“这可能是最常用的视角，但在许多方面，采取规约视角经常会更好。”这个视角要回答的问题是：“软件怎样履行自己的责任？”

视角有何用?

我们再来看看前面那个“去下一堂课教室”的例子。请注意，作为讲师的你是在概念层次上与人交流。换句话说，你告诉学生的是“你要他

们做什么”，而非“如何去做”。但是，他们如何去下一堂课的教室则是非常明确的，因为他们遵循着明确的指令，而这是在实现层次进行的。

在一个层次（概念）上交流，而在另一个层次（实现）上执行，这样请求者（讲师）就无需准确知道具体操作细节了，只需一般性——概念性地知道即可。这一点的效力可能非常大：只要概念不变，请求者就与实现细节的变化隔离开了。接下来我们来看如何描述这些概念，以及如何编写使用这些概念的程序。

1.6 面向对象范型

使用对象将责任转移到更局部的层次

面向对象范型以对象概念为中心，一切都集中在对象上。编写代码时是围绕对象而非函数进行组织的。

对象是什么？对象传统上被定义为带有方法（面向对象领域称呼函数的术语）的数据。糟糕的是，这是一种非常有局限性的对象观。稍后我会给出一个更好的对象定义（在第8章中还会谈到）。我说到对象的数据时，可能指数值和字符串这样的简单事物，也可能指其他对象。

使用对象的优点在于，可以定义自己负责自己的事物（参见表1-2）。对象天生就知道自己的类型。对象中的数据能够告诉它自己的状态如何，而对象中的代码能够使它正确工作（也就是说，做要求它做的事情）。

表1-2 对象及其责任

对　象	责　任
Student	知道自己所在的教室 知道自己下堂课的教室 从一个教室去下一个教室
Instructor	告诉学生到下堂课的教室去
Classroom	有明确地址
Direction giver	对于给定的两个教室，指出从一个教室到另一个教室的路线

在这种情况下，对象是通过寻找在问题领域中的实体而被发现的。然后再通过查看这些实体需要做些什么，为每个对象确定责任（或者称方法）。这与通过在需求中寻找名词发现对象和通过寻找动词发现方法的技术是一致的。随着所遇到问题更加复杂，我们将看到，这种技术存在很大局限性，本书中我会给出一种更好的方式。但是现在我们还要从此方式开始入手。

怎么理解对象？

理解对象的最佳方式，是将其看成“具有责任的东西”。有一条好的设计规则：对象应该自己负责自己，而且应该清楚地定义责任。这就是我之所以说“Student 对象的责任之一是知道怎样从一个教室去下一个教室”的原因。

或者，使用Fowler的视角

还可以用Martin Fowler的视角框架来观察对象：

在概念层次上，对象是一组责任；[5]

在规约层次上，对象是一组可以被其他对象或对象自己调用的方法（也称行为）；

在实现层次上，对象是代码和数据，以及它们之间的计算交互。

糟糕的是，人们对面向对象设计的教学和讨论更多的是停留在实现层次上——也就是只考虑代码和数据，对概念层次和规约层次重视很不够。然而，从后两种层次去思考对象其实也具有巨大的效力。

对象具有供其他对象使用的接口

因为对象具有责任而且自己负责自己，所以必须有办法告诉对象要做什么。还记得吗？对象含有说明自己状态的数据，还有实现必要功能的方法。对象的很多方法都将标识为可被其他对象调用。这些方法的集合就称为对象的公开接口（public interface）。

例如，在教室的例子中，我可以编写含有一个gotoNextClassroom()方法的 Student 对象。我不需要向这个方法传递任何参数，因为每个

Student对象都自己负责自己。也就是说，Student对象知道：

为了能够找到下一个教室，它需要什么；

怎样为完成这个任务获取所需的其他信息。

围绕类组织对象

刚开始，只有一种学生——普通学生需要从一个教室到另一个教室去。请注意，在我的教室（我的系统）中可能有很多这样的“普通学生”。当然可以每个学生都有一个对象对应，从而能够容易地和分别地跟踪每个学生的状态。但是，要求每个 Student 对象都有自己的一组方法，告诉它能做什么和怎样做，显然效率很低，尤其是在对所有学生而言任务都一样的时候。

一种效率更高的办法是，让所有学生与一组方法关联起来，每个学生都可以根据自己的需要使用或修改这些方法。我希望定义一个“一般学生”来包含这些公共方法的定义。然后，可以有各种各样特殊的学生，每个特殊学生都必须掌握自己的私有信息。

在面向对象术语中，这种“一般学生”被称为类（class）。类就是对对象行为的定义，它包含以下内容的完整描述：

对象所包含的数据元素；

对象能够操作的方法；

访问这些数据元素和方法的方式。

因为对象所包含的数据元素可以不同，所以同一类型的对象可以含有不同数据，但它们都具有相同的功能（如方法所定义）。

对象是类的实例

要获得一个对象时，我告诉程序需要某个类型（type，也就是对象所属的类）的一个新对象，这个新对象称为类的一个实例（instance）。创建类实例的过程称为实例化（instantiation）。

在例子中使用对象

使用面向对象方法为“去下堂课教室”的例子编写代码比以前的方法

简单多了。步骤如下所示。

1.开始控制程序。

2.实例化室中学生的集合。

3.告诉此集合，让学生去自己下堂课的教室。

4.集合让每个学生去自己下堂课的教室。

5.每个学生都：

a.找到自己下堂课的教室在哪里；

b.决定怎么去；

c.去那里。

6.完成。

抽象类型的需要

在需要加入另一个学生类型比如研究生之前，这一方式能够很好地工作。

遇到难题了。看起来我必须允许任何类型的学生（普通学生或者研究生）加入这个集合。但我面临的问题是，怎样让集合引用其元素呢？因为我是在讨论如何用代码实现，这时集合实际上将是包含某个类型对象的数组或其他容器。如果集合命名为RegularStudent（普通学生）之类，我就不能将 GraduateStudent（研究生）类型的对象放入集合。如果我说集合仅仅是一组对象，我又怎么确定其中不包含类型错误的对象（即不能“去下堂课的教室”）呢？

解决方案很直截了当。我需要一个能包容多种具体类型的一般类型。在本例中，我需要一个包含RegularStudent对象和GraduateStudent对象的Student类型。在面向对象的术语中，我们称Student类为抽象类（abstract class）[6]。

抽象类定义了一组类可以做什么

抽象类定义了其他一些相关类的行为。这些“其他”类是代表了某种特殊类型的相关行为的类。这样的类通常被称为具体类（concrete

class），因为它代表着一个概念特定的、不变的实现。

在本例中，Student就是抽象类。具体类RegularStudent和GraduateStudent则代表了两种类型的Student。RegularStudent是一种Student，GraduateStudent也是一种Student。

这种关系叫做is-a（是一个/种）关系，是我们称之为继承（inheritance）关系的一种特例。于是，我们说RegularStudent类继承自Student类。其他类似的说法还有：GraduateStudent派生自Student类，Graduate-Student特化（specialize）了Student类，或者GraduateStudent是Student的子类。

而另一方面，我们说Student类是GraduateStudent类和Regular-Student的基类，Student类泛化（generalize）了二者，或者Student类是GraduateStudent类和RegularStudent的超类（superclass）。

抽象类可以充当其他类的占位符

抽象类可以充当其他类的占位符。可以使用抽象类定义其派生类必须实现的方法。抽象类还可以包含所有派生类都能够使用的公共方法。[7]派生类是使用抽象类的默认行为还是使用自己的有所变化的行为，由派生类自己决定（这与“对象自己负责自己”的要求一致）。

这就意味着，我可以在控制程序中编写一些对象，它们的引用类型都是Student。编译器能够检查Student引用所指向的是否真的是一种Student。这种机制使我们能够实现鱼与熊掌兼得，同时获得了以下两方面的优点。

集合只需处理 Student对象（从而使 Instructor对象也只需要处理Student对象）。

但是类型检查仍然存在（只有能够“去下堂课教室”的 Student对象会包含进来）。

而且每一种Student都可以按自己的方式实现功能。

抽象类不只是不能实例化

抽象类经常被描述为“不能实例化的类”。这个定义本身没错——在实现层次上。但是局限性太大了。在概念层次上定义抽象类会更有帮助。在概念层次，抽象类就是（实现抽象类所代表的概念的）其他类的占位符。

也就是说，抽象类为我们提供了一种方法，能够给一组相关的类赋予一个名字。这使我们能够将这一组相关类看成一个概念。

在面向对象范型中，必须总是从概念、规约和实现所有三个视角层次来思考问题。

可见性

因为对象都自己负责自己，所以有很多东西不需要暴露给其他对象。前面我曾提到公开接口——可以被其他对象访问的方法的概念。在面向对象系统中，可访问性主要分为以下几种类型[8]。

公开（public）——任何对象都能够看见。

保护（protected）——只有这个类及其派生类的对象能够看见。

私有（private）——只有这个类的对象能够看见。

这就引出了封装（encapsulation）的概念。封装经常被简单地描述成“数据隐藏”。一般而言，对象不应该将内部数据成员暴露给外部世界。（也就是说，其可见性是protected或private。）

封装

但封装可不只是指数据隐藏。封装一般意味着各种隐藏。

在本例中，讲师不知道哪些是普通学生，哪些是研究生。所以学生的类型对讲师隐藏了。（也就是说，我封装了学生的类型。）在面向对象语言中，抽象类Student将隐藏从其派生的类的类型。你将在本书后面看到，这是一个非常重要的概念。

多态

另一个要理解的术语是多态（polymorphism）。

在面向对象语言中，我们经常用抽象类类型的引用来引用对象。但是，我们真正引用的是从抽象类派生的类的具体实例。

因此，当我通过抽象引用概念性地要求对象做什么时，将得到不同的行为，具体行为取决于派生对象的具体类型。“多态”这个词来源于“poly”（意为“很多”）和“morph”（意为“形态”）。因此，它的意思是“很多形态”。这个名称非常合适，因为同一个调用能够获得很多不同形态的行为。

在本例中，讲师告诉学生“去下堂课的教室”。但是，根据学生类型的不同，他们会采取不同的行为（因此出现了多态）。

面向对象术语回顾

术　语	描　述
抽象类（abstract class）	定义了一组相关类的行为

（续）

术　语	描　述
类（class）	根据对象所具有的责任定义对象的类型。责任可以分为行为和/或状态。这些分别是由方法和/或数据实现的
具体类（concrete class）	实现抽象类某一特定类型行为的类。具体类是一个概念特定、不变的实现
封装（encapsulation）	通常定义为数据隐藏，但最好将它看作任何形式的隐藏（类型、实现和设计等等）
继承（inheritance）	一个类继承另一个类，是指它接受了该类的一些或者所有性质。起始类称为基类、超类、父类或者泛化类，而继承类称为派生类、子类或者特化类
实例（instance）	类的特例（总是一个对象）。类的特殊实例或者实体。每个对象都有自己的状态。因此同一个类型（类）可以有多个对象*
实例化（instantiation）	创建类的一个实例的过程
接口（interface）	接口与类类似，但是只为其成员提供规约而不提供实现。它与只含有抽象成员的抽象类很相似。编程的时候，如果需要几个类共享公共基类中没有的一些特性，而且希望确保每个类自己实现这些特性（因为所有成员都是抽象的），就应该使用接口
视角（perspective）	观察对象有三种视角：概念视角、规约视角和实现视角。这三个不同层次的区别在理解抽象类与其派生类之间的关系上用处很大。抽象类定义了如何在概念层次上解决问题，还提供了与任何派生对象通信的规约。每个派生类都按需要提供特定的实现
多态（polymorphism）	能够用一种方式引用一个类的不同派生类，但获得的行为对应于所引用的派生类

* 有些面向对象分析人员说万事万物皆对象：类是对象，实例也是对象。这在技术上可能是正确的，但是却成了混淆和发生争议的地方。本书所称的对象是类的实例。

1.7 面向对象程序设计实践

新实例

我们再次考察一下本章开始讨论的形状实例。怎样用面向对象的方式实现它呢？请记住，我们必须完成以下任务。

1.在数据库中找到形状列表。

2.打开形状列表。

3.按某种规则将列表排序。

4.在显示器上显示各个形状。

为了用面向对象方式解决这个问题，我需要定义一些对象和这些对象具有的责任。

在Shape程序中使用对象

所需要的对象如下表所示。

类	责任（方法）
ShapeDataBase	getCollection——获取指定形状的集合
Shape（抽象类）	display——为形状定义接口 getX——返回形状的 x 坐标（用于排序） getY——返回形状的 y 坐标（用于排序）
Square（派生自 Shape 类）	display——显示一个正方形（该类对象代表的形状）
Circle（派生自 Shape 类）	display——显示一个圆形（该类对象代表的形状）
Collection	display——显示所存放的所有形状 sort——对形状集合排序
Display	drawLine——在屏幕上画一条线 drawCircle——在屏幕上画一个圆

运行程序

现在主程序的步骤应该与下面给出的类似。

1.主程序创建一个数据库（ShapeDataBase）对象的实例。

2.主程序要求数据库对象找到我感兴趣的一组形状，然后实例化一个保存这些形状的 Collection 对象（实际上，它还将实例化Collection对象中存放的Circle对象和Square对象）。

3.主程序要求Collection对象将所存放的形状排序。

4.主程序要求Collection对象显示形状。

5.Collection对象要求所存放的所有形状显示自己。

6.每个形状根据形状种类显示自己（使用Display对象）。

为什么这有助于应对新需求

我们来看这个方案怎么会有助于我们应对新的需求（请记住，需求总在变化）。例如，考虑如下的新需求。

增加新种类的形状（例如三角形）。为了引入一种新的形状，只需两步：

创建Shape类的一个新的派生类，来定义这个新形状；

在新的派生类中，实现与该形状对应的display方法。

修改排序算法。为了修改形状排序方法，只需一步：

修改Collection的sort方法。这样所有形状都将使用新算法。

结论：面向对象方法有效地限制了需求变更所带来的影响。

再谈封装

封装有几个优点。“对用户隐藏”这一事实直接蕴涵了以下优点。

使用更容易，因为用户不需要再操心实现问题了。

可以在不考虑调用者的情况下修改实现。（因为调用者从一开始就不知道对象是如何实现的，它们之间不应该存在任何依赖关系。请记住，在维护中时间往往花在了解和留心这些依赖关系上，而不是实际添加新功能。）

其他对象对该对象内部是未知的——这些外部对象往往用来帮助实现该对象接口所指定的功能。

优点：减少副作用

最后，考虑功能改变时引起的不良副作用问题。这种bug通过封装有效地解决了。对象内部对于其他对象是未知的。如果使用封装，并遵循“对象自己负责自己”的策略，那么唯一能影响对象的办法就是调用该对象的方法。对象的数据和实现其责任的方式都与其他对象所带来的变化屏蔽开来。

封装拯救了我们
对象对自己行为所负的责任越多，控制程序需要负的责任就越少。 封装使对象内部行为的变化对其他对象变得透明了[9]。 封装有助于防止不良副作用。 值得注意的是封装与耦合的关系。封装什么东西时，必然将使其耦合变松。隐藏实现（即封装它们）有助于松耦合。

1.8 特殊对象方法

创建和销毁

我已经讨论了可能被其他对象或对象自己调用的方法，但是当对象创建时到底发生了什么事情？当它消亡时又发生了什么？如果对象应该是自成一体的单位，那么它自己包含处理这些情况的方法，将是一个不错的主意。

这些特殊方法事实上确实存在，它们就是构造函数（constructor）和析构函数（destructor，或者终结方法，finalizer）。

构造函数负责初始化或创建一个对象

构造函数是对象创建时自动调用的一个特殊方法，它的目的是处理对象起始时的工作，这是对象“自己负责自己”所要求的。构造函数是一个进行初始化、设置默认信息、设定与其他对象关系或创建一个明确的对象所需的其他工作的天然场所。所有面向对象语言都会在创建对象时查找并执行相应的构造函数。

通过正确使用构造函数，可以更容易消除（或者至少最大程度地减少）未初始化变量，这种错误通常源于开发者的粗心大意。如果代码中有一个固定且一致的地方（即对象的构造函数）进行所有的初始化工作，可以更容易地确保初始化。未初始化变量所引起的错误很容易改正，但很难发现，因此这种约定（以及构造函数的自动调用）能够提高程序员的效率。

析构函数（终结方法）在对象不再需要时（已被删除时）将其清除

大多数面向对象语言都提供了对象不再存在时清除该对象的方式。在C++和C#中称之为析构函数，在Java中称之为终结方法。本书中，我将采用通用术语析构函数称呼这一概念。

所有面向对象语言都会在对象删除时查找并执行相应的析构函数。与构造函数一样，析构函数的使用也是对象“自己负责自己”所要求的。

析构函数通常用于在对象不再需要时释放资源。因为Java有垃圾收集机制（对象不再使用时自动清除）[10]，析构函数在Java中不像

C++中那么重要。在C++中，由对象的析构函数负责销毁只由这个对象使用的其他对象是很常见的。

1.9 小结

本章内容

本章中我说明了面向对象技术是怎样帮助我们最大程度地减少系统需求变更带来的影响，以及面向对象与功能分解的异同。

我还讨论了面向对象程序设计的许多基本概念，介绍和描述了主要术语。表1-3总结了这些概念，表1-4总结了面向对象程序设计的主要术语。

表1-3 本章介绍的概念

概　念	回　顾
功能分解	结构化程序员总是使用功能分解进行程序设计。功能分解是将一个问题逐渐分解为更小功能的方法，每个函数都分解到可管理为止
需求变更	需求变更是开发过程中与生俱来的。与其向用户或者自己抱怨“获得理想而且完整的需求看来是不可能完成的任务”，不如使用更有效的开发方法，应对需求变更
对象	对象是由其责任定义的。对象能够自己负责自己，从而简化了使用对象的控制程序的任务
构造函数和析构函数	对象具有在创建和销毁自己时自动调用的特殊方法。这些特殊方法是： ❑ 构造函数：初始化或创建对象 ❑ 析构函数：在对象删除时清除对象 所有面向对象语言都使用构造函数和析构函数辅助对对象的管理

表1-4 面向对象术语

术　语	定　义
抽象类	定义了概念上相似的一组类的方法和公共属性。抽象类不能实例化
属性	与对象关联的数据（也称为数据成员，data member）
类	对象的蓝图——为其类型的对象定义方法和数据

（续）

术　语	定　义
构造函数	在创建对象时调用的特殊方法
派生类	从基类特化的类，包含基类所有的属性和方法，但还可能包含其他属性或不同的方法实现
析构函数	在销毁对象时调用的特殊方法
封装	任何形式的隐藏。对象封装其数据，抽象类封装其派生的具体类
功能分解	一种分析方法，将问题逐步分解成小的功能
继承	一种特化类的方式，用于将派生类与其基类联系起来
实例	类的特定对象
实例化	创建类的一个实例的过程
成员	类的数据或方法
方法	与对象关联的例程
对象	具有责任的实体。一个特殊的、自成一体的容器，包含数据和操作数据的方法。对象的数据对于外部对象是受保护的
多态	相关的对象根据其具体类型实现方法的能力
超类	其他的类从中派生的类，包含所有派生类都要使用的主要数据和方法的定义（方法可能改写）

复习题

简答题

1.叙述功能分解中使用的基本方法。

2.导致需求变更的三个原因是什么？

3.我提倡用责任而不是功能来思考。这意味着什么呢？请举出一个例子。

4.给出耦合和内聚的定义。什么是紧耦合？

5.对象接口的目的是什么？

6.给出类实例的定义。

7.类是一个对象行为的完整定义。这句话说明了对象的哪三个方面？

8.抽象类的作用是什么？

9.对象可能具有的三种主要可访问性[11]是什么？

10.给出封装的定义，并举出一个行为封装的例子。

11.给出多态的定义，并举出一个多态的例子。

12.观察对象的三种视角是什么？

阐述题

1.有时候，程序员使用“模块”来隔离不同区域的代码。这是应对需求变更的有效方式吗？为什么？

2.将抽象类定义为不能实例化的类局限性很大，为什么呢？抽象类更好的（或者至少，另一种）理解方式是什么？

3.行为的封装是怎样帮助限制需求变更带来的影响的？它又怎样挽救程序员免于无意导致的副作用？

4.接口怎样有助于保护对象不受其他对象变化的影响？

5.在一个系统中要使用教室作为描述对象。请从概念视角描述教室。

观点与应用题

1.需求变更是系统开发人员所面临的最大挑战之一。请从你自己的亲身经历中找出一个支持这一说法的例子。

2.功能分解方法在遇到需求变更时存在本质上的弱点。你同意这种说法吗？为什么？

3.你认为应对需求变更的最佳方法是什么？

第2章 UML

2.1 概览

本章内容

本章将简单概述UML（统一建模语言），这是面向对象界主要使用的一种建模语言。如果你还不知道 UML，阅读本章将使你具备阅读本书模型图所需的最低限度的知识。

本章中，我们将：

叙述“什么是UML”和“为什么使用UML”；

阐述本书中的基本UML图，即

类图；

交互图。

2.2 什么是UML

UML提供了多种建模图

UML 是一种用来创建程序模型的图形语言（即带有语意的一种图形记号）。在此上下文中，术语“程序模型”指的是程序的图形表示，可以说明代码中对象之间的关系。

UML 中有好几种不同的图——有些用于分析，有些用于设计，还有些用于实现 [更准确地说，是用于部署（deployment），也就是代码的发布（distribution）]。（参见表 2-1）根据图的目的不同，每个图都说明了不同实体集合之间的关系。

表2-1 UML图及其用途

当你在……	所使用的 UML 图
分析阶段	用例图，所涉及的是与系统（也就是用户和其他的系统）之间交互的实体，以及需要实现的功能点 活动图，关注的是问题领域[人和其他主体（agent）工作 的实际空间，程序的主题领域]的工作流，而不是程序的逻辑流 请注意：因为本书主要关注的是设计，所以不会讨论用例图或活动图

（续）

当你在……	所使用的 UML 图
观察对象的交互	交互图，说明了特定对象如何互相交互。因为它们处理的都是具体情况而不是一般情况，所以在检查需求和设计时都很有用。最常见的一种交互图是顺序图
设计阶段	类图，详细描述了类之间的关系
观察对象所处状态不同时行为的差异	状态图，详细描述了对象可能所处的不同状态以及在这些状态之间的转换
配置阶段	部署图，说明了如何部署不同模块。本书中不会谈到部署图

2.3 为什么使用UML

主要用于交流

UML　主要是用来交流的——与我自己、与我的小组成员、与我的客户。在软件开发领域中糟糕的（不完整的或者不准确的）需求无处不在，而UML为我们提供了提高需求质量的工具。

有利于清晰

UML　提供了一种方法，可以用来确定我对系统的理解是否与其他人相同。因为系统非常复杂，有许多不同种类的信息需要传递，所以UML提供了许多不同的图专门表示不同种类的信息。

有利于精确

要认识到UML的价值，有一个简单的办法：回忆最近参加的几次设计评审。如果在某次评审中，某人在不使用UML等建模语言的情况下开始谈起自己的代码并描述它，几乎能够肯定他的发言将含混难懂，而且不必要地冗长。UML不仅仅是描述面向对象设计的上佳方法，它还使设计人员能够仔细考虑其设计中类之间的关系（因为必须将设计写

下来）[12]。

2.4 类图

基本的建模图

最基本的UML图是类图。它不仅描述了类，而且说明了类之间的关系。这些关系可能有以下这些类型。

当一个类是“一种”另一个类时：is-a（是一种/一个）关系。

当两个类之间存在关联时：

一个类“包含”另一个类：has-a（拥有一个）关系；

一个类“使用”另一个类：use-a（使用一个）关系；

一个类“创建”另一类。

这些类型还有一些变体。比如，说“什么东西包含另一个东西”时，我们可能是指：

被包含者是包含者的一部分（比如汽车中的发动机）；

有一个集合，集合中东西可以独立存在（比如机场上的飞机）。

表示类信息的不同方法

第一种情况称为组合（composition），第二种情况称为聚集（aggrega-tion）[13]。

图2-1说明了重要的几点。首先，矩形表示一个类。在UML中，我可以表示最多三个方面的类的信息：

类名；

类的数据成员；

类的方法（函数）。

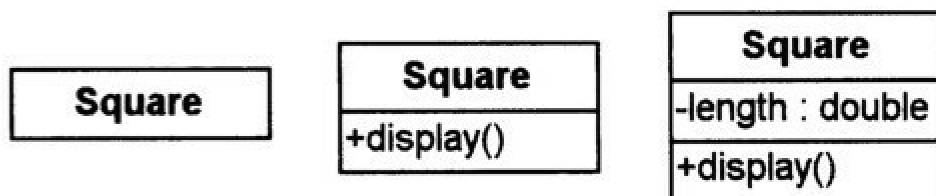


图2-1 类图——三种变体

表示类的信息有三种不同方式。

最左边的矩形只显示了类名。在不需要更详细信息时，可以使用类的这种表示形式。

中间的矩形显示了类名和类的方法。在本例中，Square类[14]有一个display方法。display（方法名）前的加号（+）表示此方法是公开的——也就是说，不属于此类的其他对象也可以调用。

最右边的矩形除显示了前面的信息（类名和类的方法）之外，还显示了类的数据成员。在本例中，数据成员length（它是double类型的）前的减号（–）表明这个数据成员的值是私有的，也就是说，除了它所属的对象外，它对其他对象都是不可见的。[15]

表示访问权限的**UML**记号

你可以控制类的数据成员和方法成员的可访问性，也可以用 UML标记所需要的每个成员的可访问性。大多数面向对象语言中都有如下三种最常见的可访问性。

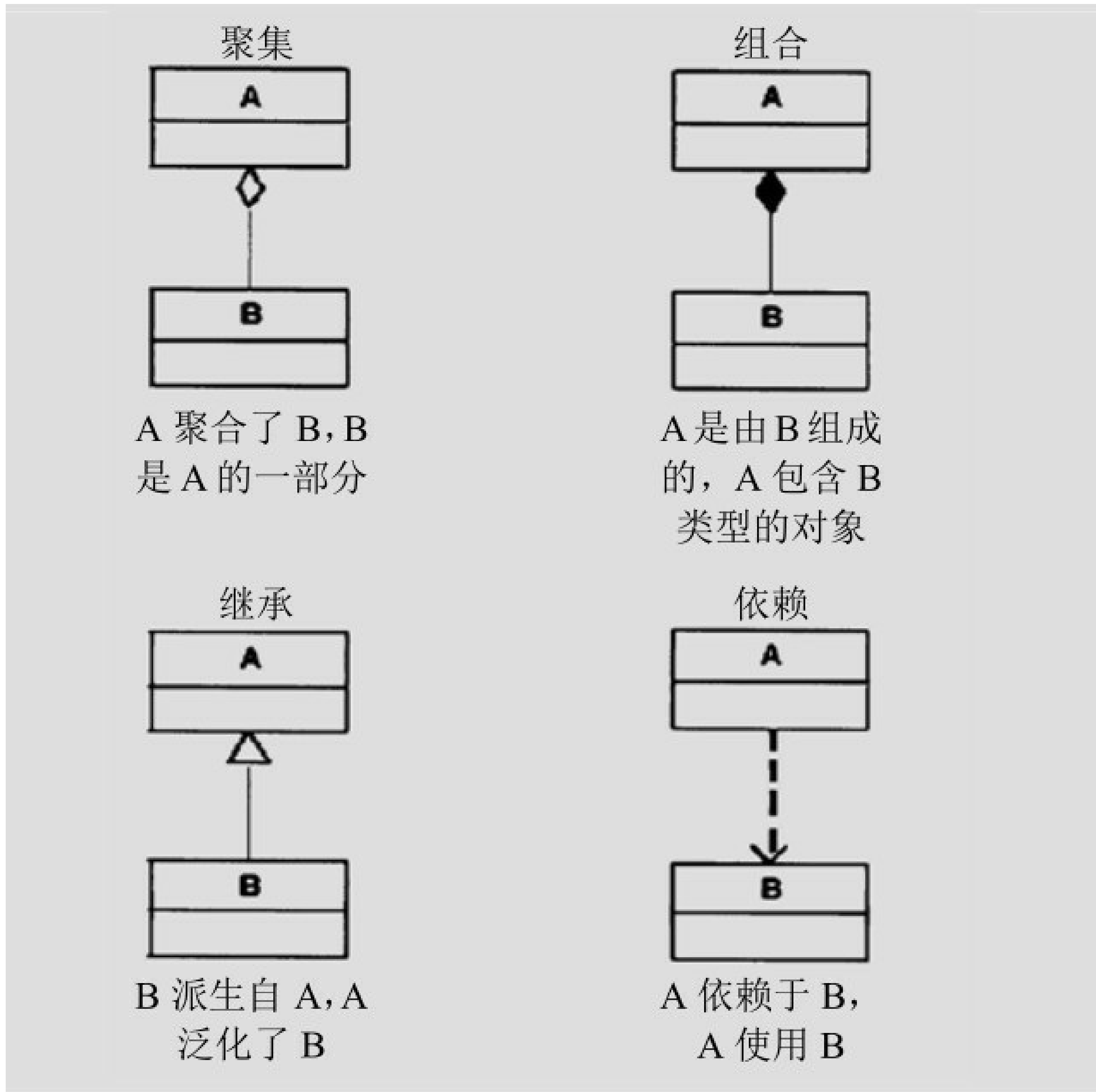
公开——用一个加号（+）标记。意味着所有对象都可以访问这个数据或方法。

保护——用一个“井”号（#）标记。意味着只有该类及其所有派生类（包括其派生类的派生类）可以访问这个数据或方法。

私有——用减号（–）标记。意味着只有该类的方法可以访问这个数据或方法。（请注意：某些语言进一步将其限制为特定对象。）

表示关系的**UML**记号

表示关系的UML记号有如下四种：[16]



类图还可以表示关系

类图还可以表示不同类之间的关系。图2-2显示了Shape类和它的几

个派生类之间的关系。

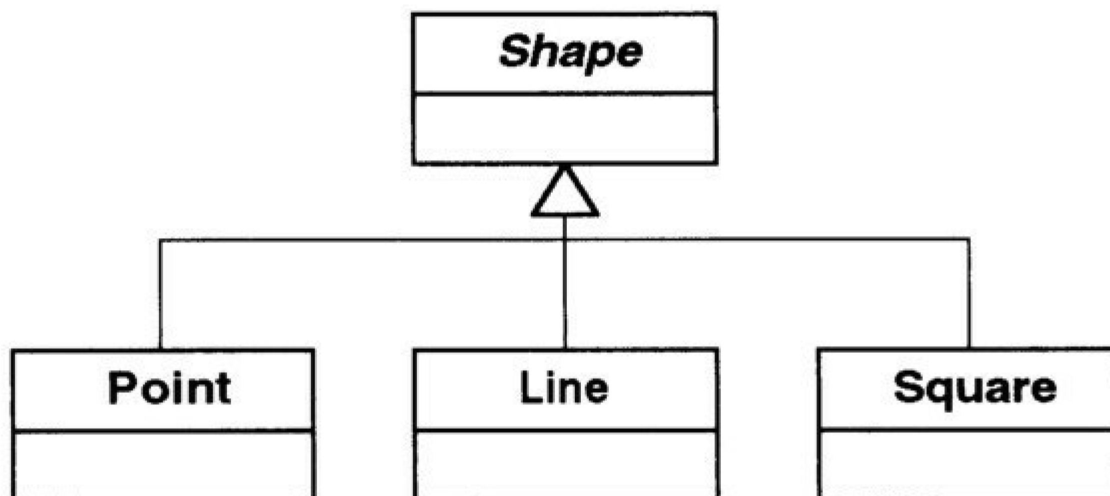


图2-2 显示了is-a关系的类图

表示is-a关系

图2-2说明了几件事。首先，Shape类下面的箭头的意思是：指向Shape的那些类派生自Shape类。而且，Shape类的名字是用斜体表示的，说明它是一个抽象类。抽象类就是用来为其派生类定义接口而且存放这些派生类公共数据和方法的类。接口可以看作是没有公共数据和方法的抽象类——它只用来作为一种“为要实现它的那些类的方法进行定义”的方式而已。[17]

表示has-a关系

如前所述，有两种不同的has-a关系。一个对象可以拥有另一个对象，其中被包含的对象是包含对象的一部分——或者不是。在图2-3中，我表示出Airport“拥有”Aircraft。Aircraft并不是Airport的一部分，但仍然可以说Airport拥有Aircraft，这种关系称为聚集。

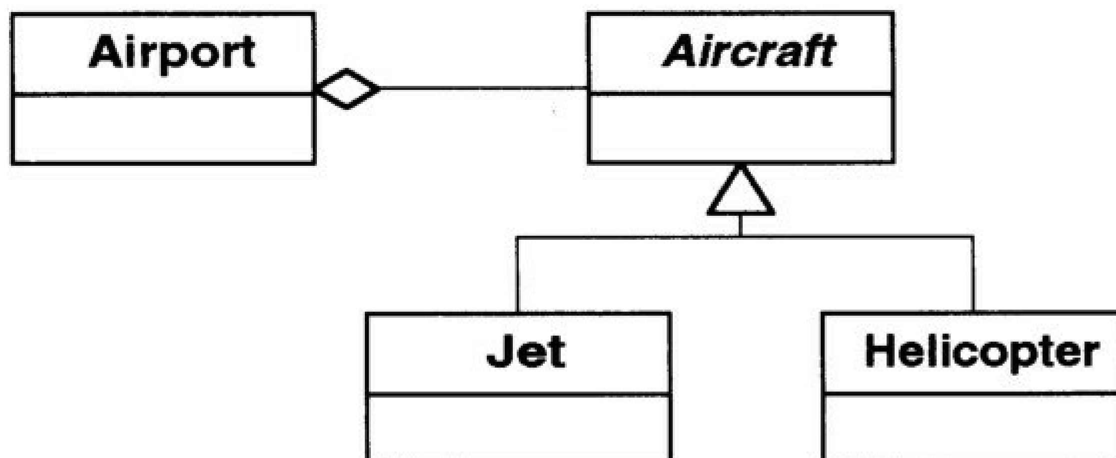


图2-3 显示了has-a关系的类图

在此图中，我还表示了Aircraft要么是Jet（喷气式飞机），要么是聚集Helicopter（直升飞机）。可以看出Aircraft类是一个抽象类或者接口[18]，因为它的名字是用斜体表示的。也就是说，Airport可以拥有Jet或Helicopter，但它是以相同方式对待它们的（当作Aircraft）。Airport类右边的空心（未填充的）菱形表示聚集关系。

组合

另一种has-a关系是包含，被包含对象是包含对象的一部分，这种关系也称为组合。

组合和使用

图2-4显示了Car（轿车）拥有Tire（轮胎），后者是它的一部分（也就是说，Car由Tire和其他东西组成），这种has-a关系，称为组合关系（composition），用实心菱形表示。此图上还显示了Car使用了GasStation（加油站）类，这种使用关系用带箭头的虚线表示，也称依赖关系（dependency relationship）。

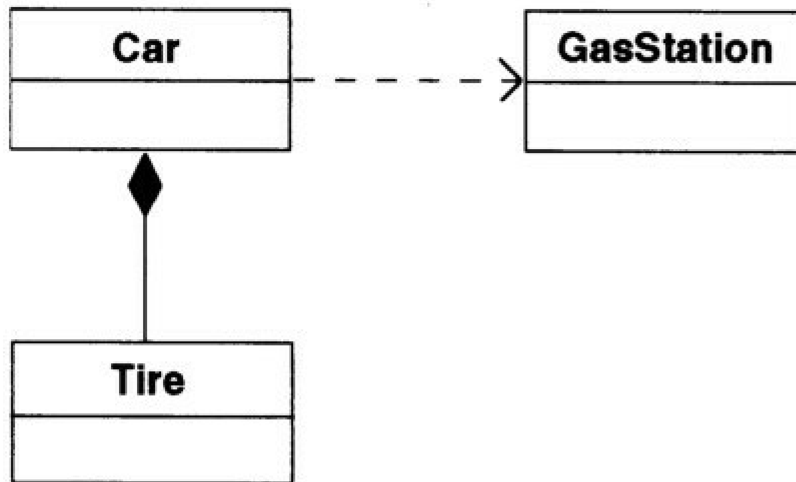


图2-4 显示了组合及使用关系的类图

组合与聚集的异同

组合和聚集都有“一个对象包含一个或多个对象”的意思，但是，组合意味着“被包含对象是包含对象的一部分”，而聚集意味着被包含对象更像是一个集合。我们可以认为组合是一种非共享的关联，被包含对象的生存周期由包含对象控制。适当使用构造函数和析构函数在这里有助于对象的创建和销毁过程。

UML中的注释

在图2-5中有一个新记号：注释。含有“空心菱形表示聚集”信息的方框就是注释。注释记号看上去好像是右角折起的纸。经常能够看到注释通过一条线与特定的类连接起来，表示它只与此类有关。

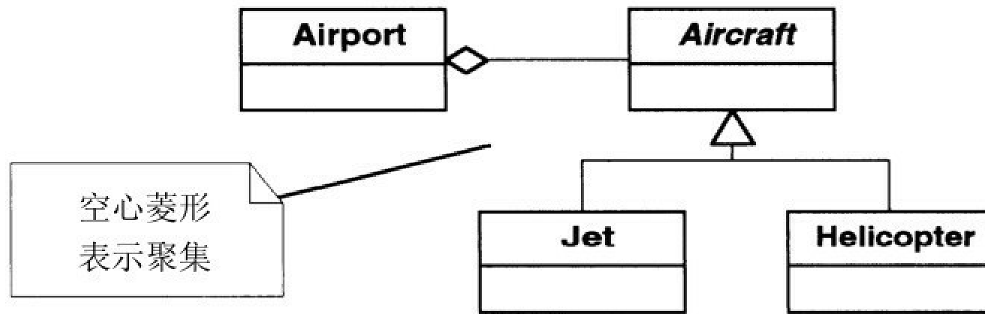


图2-5 带注释的类图

表示另一个对象所拥有的东西的数量

类图表示的是类之间的关系，但是，对于组合和聚集来说，这两种关系更加关注该类型的具体对象。比如，Airport对象拥有Aircraft对象，但是更具体地说，是特定的机场拥有特定的飞机。于是问题出现了——“一个机场可以拥有多少架飞机呢？”这称为关系的重数（cardinality）。图2-6和图2-7说明了这一点。

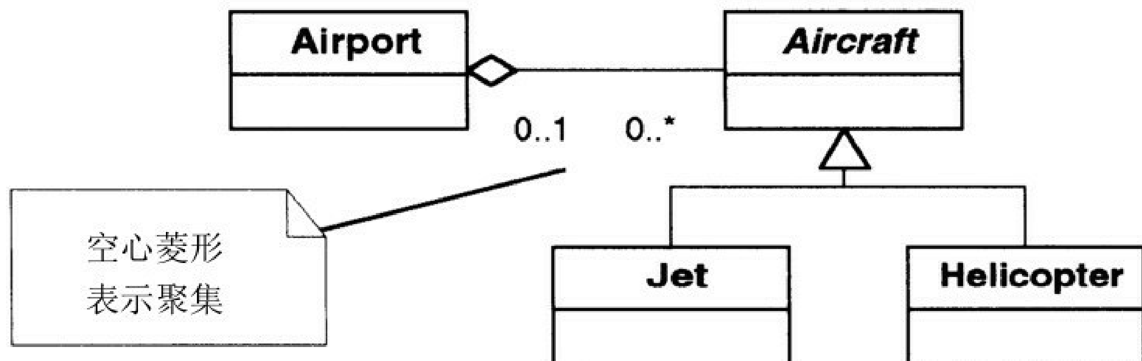


图2-6 Airport-Aircraft关系的重数

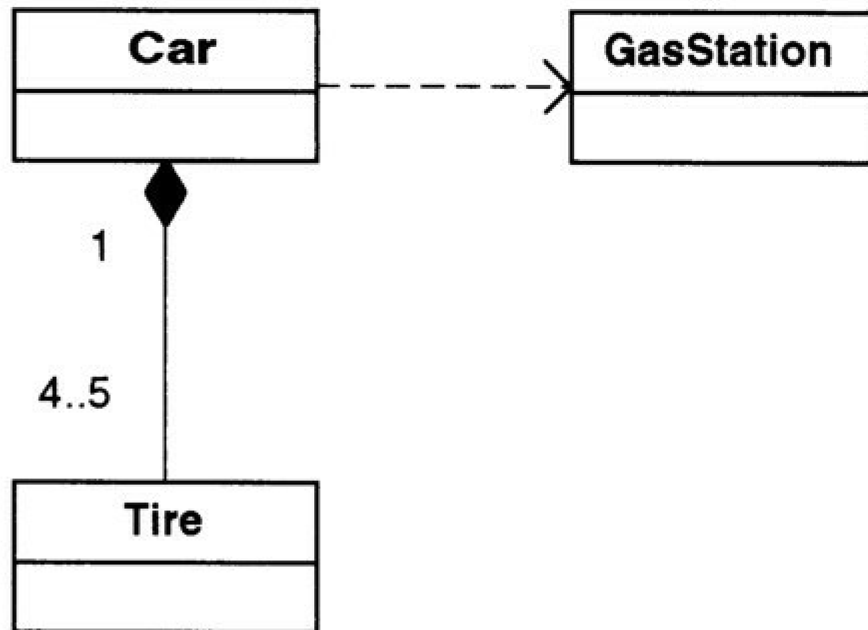


图2-7 Car-Tire关系的重数

重数

图2-6告诉我们，对于一个Airport对象，它可以拥有从0到任意数量（此处用星号表示，但有时候也可以用字母“n”）的Aircraft对象。Airport类旁的“0..1”意味着：对于一个 Aircraft 对象，它可以被 0个或1个 Airport对象包含。（0 表示它可以在空中飞行，不属于任何机场）。

重数续

图2-7告诉我们，对于一个Car对象，它可以拥有4个或5个Tire对象（有或没有备胎），轮胎则只能装在一辆轿车上。我曾听一些人说，如果未指定重数，就意味着只有一个对象，这种想法是不正确的。如果未指定重数，对于对象的数量不应该做任何假设。

虚线表示依赖

和前面一样，图2-7中显示的Car和GasStation之间的虚线表示两者之间存在依赖。UML 用带虚线的箭头表示两个模型元素之间的语义关系（意义）。

2.5 交互图

交互图

类图可以表示类之间的静态关系，换句话说，类图不能表示任何活动。虽然这非常有用，但有时候我需要表示这些类实例化的对象是如何实际地一起工作的。

表示对象间如何交互的UML图称为交互图（interaction diagram）。最常用的交互图是顺序图，如图2-8所示。

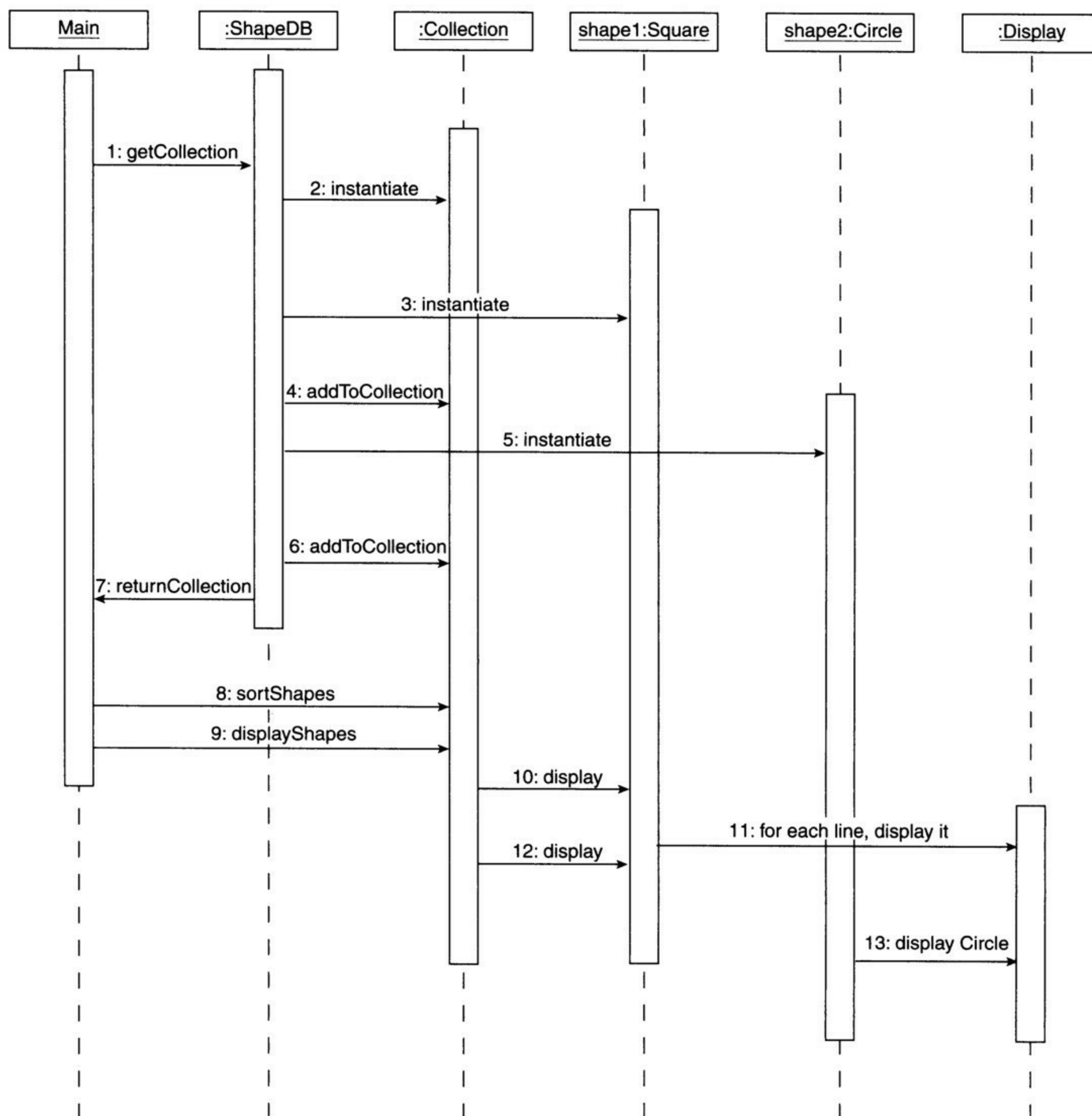


图2-8 Shapes程序的顺序图

顺序图应该从顶到底地阅读，如下所述。

如何阅读顺序图

最上面的每个矩形都代表一个特定的对象。虽然许多矩形中有类名，但请注意在类名前有一个冒号。一些矩形还有其他名字——例如shape1:Square。

垂直线代表对象的生命线。糟糕的是，大多数UML绘图程序不支持这一点，只能绘制从顶到底的线，因此并不清楚对象实际上什么时候开始存在。

我用这些垂直线之间的水平线表示对象互相发送消息[19]。

有时候返回值和/或对象会明确表示出来，而有时候只是表示它们要返回。

例如，在图2-8中，

在最上面可以看见Main向ShapeDB对象（这个对象还没有名字）发送了一个“获取形状集合”的消息。

在收到“获取形状集合”的请求之后，ShapeDB对象将：

实例化一个Collection对象；

实例化一个Square对象；

在集合中添加Square对象；

实例化一个Circle对象；

在集合中添加Circle对象；

将集合返回给调用例程（Main）。

其余操作也可以通过这种从顶到底的方式读图来了解，这种图称为顺序图（sequence diagram），因为它描述了操作的顺序。

“对象:类”记号

有些UML图中，需要用派生对象的类来表示该对象。可以通过用冒号连接二者来实现这一点。在图2-8中，我用 shape1:Square 表示从Square类实例化的shape1对象。

2.6 小结

本章内容

UML 既能够充实设计，又能够用于设计的交流。不要太担心要“正确地”画图。要考虑的是什么方式最有利于交流设计中的概念。换句话说，

如果你认为有什么东西需要说，可以用注释来表达。

如果你对一个图标或符号不太确定，必须查手册才能确定其意义，还是加一条注释来解释。毕竟，其他人有可能也不清楚它的意义。

清晰为好。

当然，这也意味着你应该以规范的方式使用 UML——那样无法正常交流。在画图的时候，只考虑要传达的思想即可。

复习题

简答题

1.is-a关系和has-a关系之间的区别是什么？两种“关联”关系又是什么？

2.在类图中，类是用方框表示的，可以有三部分。请描述这三部分。

3.给出重数的定义。

4.顺序图的用途是什么？

阐述题

1.给出is-a关系和两种“关联”关系的例子。对这些例子：

(1) 在类图中画出；

(2) 在类图中显示重数。

2.图2-8是一个顺序图。此图中显示了多少步骤？显示了多少对象，都是哪些对象？

3.当对象互相交流时，为什么说“发送消息”比“调用操作”更合适？

观点与应用题

一个顺序图上应该显示多少步？

---

[1].McConnell S.，Code Complete:A Practical Handbook of Software Construction，Redmond: Microsoft Press，1993，p.81。（请注意，这些术语并不是McConnell发明的，发明者是Ed Yourdon 和 Constantine。我们只是碰巧更喜欢McConnell的定义而已。）

[2].McConnell S.，Code Complete:A Practical Handbook of Software Construction，Redmond: Microsoft Press，1993，p.81。（请注意，这些术语并不是McConnell发明的，发明者是Ed Yourdon 和 Constantine。我们只是碰巧更喜欢McConnell的定义而已。），第87页。

[3].应该指学校中类似于JavaOne的技术大会，有很多课程和讲座同时在不同地点开设。——译者注

[4].Fowler M.和Scott K.，UML Distilled:A Brief Guide to the Standard Object Modeling Language，Second Edition，Boston:Addison-Wesley，1999，pp.51-52。

[5].这里比较粗略地套用了Bertrand Meyer在Object-Oriented Software Construction（Upper Saddle River，N.J.: Prentice Hall，1997，p.331）中概述的“按约定设计”（design by contract）的概念。

[6].有几种语言中接口也能如此。本章中提到抽象类时，可以假定我编写的是抽象类或者接口。

[7].抽象类和接口之间有一点不同。接口只定义一组类能够做什么，而并不实现默认行为。

[8].不同的语言经常有其他类型的可访问性，但是，本质上都是这三种的变种而已。

[9].即不可见了。——译者注

[10]..NET语言亦然，包括C#、VB.NET和C++/CLI等。——译者注

[11].即可见性。——译者注

[12].有些敏捷方法专家相信，各种书面的文档都应该避免，除非绝对需要。当然，许多开发人员对UML的使用的确有些过分，而且所生成的文档实际上是阻碍而不是促进了交流。但是，只要正确地使用，UML还是能很好地促进交流的，即使在使用“结对编程（paired programming）”时，设计概念在概念层次描述通常也比在代码（即实现）层次描述更好。换句话说，应该努力同时做到“尽可能最简”和“尽可能最好”。

[13].Gamma、Helm、Johnson 和 Vlissides 的《设计模式》一书中将第一种情况称为“聚集”，而将第二种情况称为“组合”（《设计模式》一书中aggregation的确相当于本书中的组合概念，但是该书中的composition则是指对象组合，与继承相对，和本书中的聚集没有关系。——译者注）——正好与UML相反。但是，该书完成于UML标准最终定案之前，事实上书中所给出的定义是与UML一致的。这也说明了开发UML的动机。在UML出现之前已经有好几种各不相同的建模语言，每种都有自己的记号和术语。

[14].类名在文字中引用时，使用Courier字体表示。

[15].在一些编程语言中，同类型的对象可以相互共享私有数据。

[16].原书此段文字与“表示访问权限的UML记号”中的一段文字相同，估计是作者的失误。——译者注

[17].我知道自己两次使用接口一词，表示的是不同的含义。但是别为此骂我。我也希望对Java和C＃的关键字——interface使用另一个名字呢。

[18].为了简明起见，我不再继续写“抽象类或者接口”了。往后我所称的“抽象类”，均可以视同为“抽象类或者接口”。

[19].当对象互相“交谈”时，我们称之为“发送消息”。你需要给一个对象发送请求，让它进行某种操作，而不是告诉其他对象做什么，其他对象会负责搞清楚如何去做。转移责任是面向对象程序设计基本原则之一。这与过程式程序设计完全不同，在后者情况下，你必须控制下一步做什么，因此可能“调用另一个对象的方法”或者“调用操作”。

第二部分 传统面向对象设计的局限

概览

本部分内容

在本部分中，我将用标准面向对象方法解决一个实际问题。这个问题是我刚开始学习设计模式时碰到的。

章 讨论的主题

3 对代码灵活性要求很高的问题

对CAD/CAM问题的描述：从一个大型CAD/CAM（计算机辅助设计/计算机辅助制造）数据库中提取信息，提供给一个复杂而且昂贵的分析程序。

因为CAD/CAM系统一直在演进，所以这个问题要求代码非常灵活。

4 标准的面向对象解决方案

我对CAD/CAM问题的第一个解决方案，使用了标准的面向对象方法。

在实际着手解决这个问题时，我还没有领会设计模式之后隐藏的那些原则的精髓，所以，我的最初解决方案过分依赖于继承机制。设计过程很轻松，最初的方案也可以正常工作，但最终特殊情况太多了。

我的方案问题明显——难以维护、灵活性不够。这些恰恰是我希望能够通过面向对象设计避免的问题。

在第四部分的第13章中我们会再次谈到这个问题。我将通过设计模式再次解决它，从而将应用程序的架构及其实现细节协调地组织起来。

这样将设计出一个维护性和灵活性都好得多的解决方案。

本部分意义

阅读本部分非常重要，因为它说明了传统面向对象设计中的一个典型问题——没有必要的继承层次结构，这种结构往往是紧耦合而且低内聚的。

第3章 对代码灵活性要求很高的问题

3.1 概览

本章内容

本章将概述我们要解决的一个问题：从一个大型CAD/CAM[1]数据库中提取信息，提供一个复杂而且昂贵的分析程序。因为CAD/CAM系统还在继续发展改进，所以这个问题对代码的灵活性要求很高。

本章中，我将分析这个CAD/CAM问题，给出问题领域的术语表，以及问题的重要特征。

3.2 提取CAD/CAM系统的信息

问题：为专家系统提取信息

现在我将回顾自己以前的设计，正是它使我开始逐步获得本书中的领悟。

那时我正在为一个设计中心做技术支持，这个设计中心的工程师们使用一个CAD/CAM系统制作金属板材零件图。零件如图3-1所示。

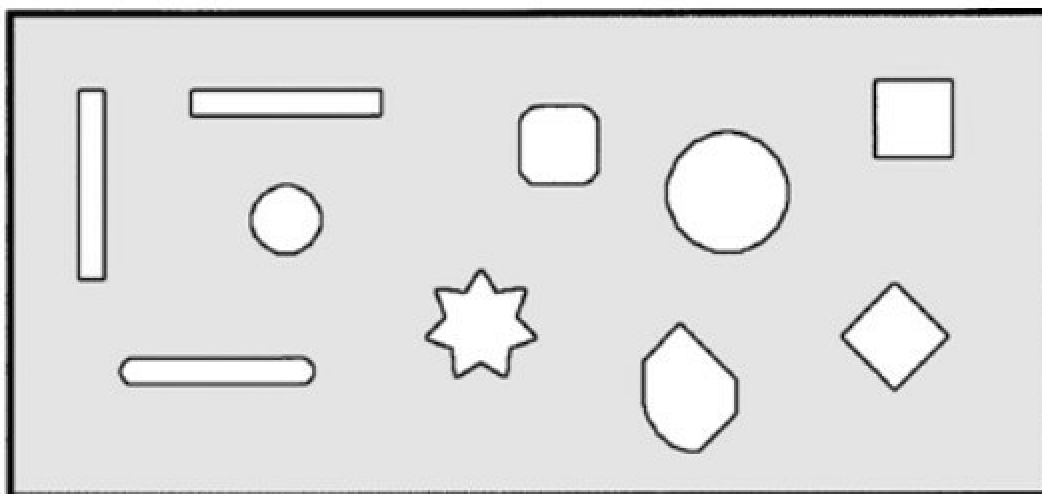


图3-1 金属板零件示例

我的任务是编写计算机软件工具从CAD/CAM系统中提取信息，然后有一个专家系统用这些信息来控制零件的制造流程。因为这个专家系统比较难于修改，而且生命期将比当前版本的CAD/CAM系统还要长，所以我想要编写的信息提取工具应该很容易适应CAD/CAM系统的新版本。

什么是专家系统

专家系统是一种特殊的计算机系统，它采用人类专家总结出的规则进行自动化决策。开发专家系统分两步。首先，获取专家用来进行决策、完成任务的规则，并对其进行建模。其次，在计算机系统中实现这些规则，此步骤通常会使用某种商用专家系统工具。对于分析人员来说，第一步骤的任务比第二步骤的要困难得多。

3.3 了解专业术语

术语：区分金属板材上切割出的各种形状

分析中的第一个任务，就是要了解问题领域中用户和专家使用的专业术语。其中最重要的，是用来描述金属板材切口的尺寸和几何特征的术语。

如图3-1所示，一块金属板材能够切割成特定的尺寸，其内部切割出各种形状，专家们统称这些切口为“部件”（feature）。一块金属板材可以用“外形尺寸”和“内部所含部件”完全规定。

表3-1列出了金属板材上可能出现的各种部件类型，这就是系统必须处理的形状。

表3-1 金属板材上出现的形状

形　状	描　述
沟槽（slot）	金属板材上宽度固定、两端为方角或圆角的直切口。沟槽可能朝向任何角度，通常是用铣刀切成的。图 3-1 左部有 3 条沟槽：1 条垂直，其他水平
孔（hole）	金属板材上的圆形切口。通常它们是用不同宽度的钻头钻出来的。图 3-1 左部 3 条沟槽间有一个孔，右部有一个更大的孔
方切口（cutout）	带有方角或圆角的正方形切口。是由高压冲床以巨大冲力在金属上冲出来的。图 3-1 中有 3 个方切口，右下角的一个倾斜 45°
特殊形状（special）	沟槽、孔、方切口之外的其他预制形状。这时，需要制造特殊冲头，进行快速冲压。电源插座是一种常见的“特殊”切口。图 3-1 中的星形切口也是一种特殊形状切口
不规则（irregular）	其他形状。是用几种工具组合起来切割而成的。图 3-1 右下部的五边不规则形状就是一个不规则形状切口

更多术语

CAD/CAM专家们还使用其他重要术语，我们也需要理解，如表3-2所示。

表3-2 CAD/CAM术语

术　语	描　述
几何特征（geometry）	金属板材的外观描述：每个部件的位置和尺寸，金属板材的外形
零件（part）	金属板材本身。我需要能够存储每个零件的几何特征
模型（model）或数据集（dataset）	CAD/CAM 数据库中存储零件几何特征的记录集
数控机床（NC machine）和数控集（NC set）	一种特殊的制造机械，能够使用计算机程序控制各种切削头切割金属。通常，需要给控制程序输入零件的几何特征。组成计算机程序的命令称为数控集

3.4 问题描述

系统任务的高层描述

我要设计的程序，应该使专家系统可以打开并读取一个模型，其中包含我要分析的零件的几何特征，然后生成命令，让数控机床制造金属板材。

在本例中，我只关心金属板材零件。但是，这个CAD/CAM系统还可以处理许多其他种类的零件。

在高层次上，我希望系统执行以下步骤。

1.分析金属板材。

2.根据板材所含部件，确定制造方式。

3.生成制造设备可读的指令集。这种指令集称为数控集。

4.在需要制造某个零件时，就将对应指令提供给制造设备。

专家系统的任务绝非无足轻重

这里编程任务的难点在于：我不能只是从模型中提取部件信息，并生成数控集命令。要用什么类型的命令和命令的顺序，都取决于部件及其与其他部件的关系。

例如，假定有一个形状由几个部件组成：带两个沟槽的方切口。其中一条沟槽与方切口垂直，另一条与方切口水平，如图3-2所示。

认识到实际上左图的形状由右图中三个部件组成，是非常重要的。这是因为使用该 CAD/CAM 系统的工程师通常会按已有的规则部件思考

更复杂的形状，因为他们知道这样可以更快地制造零件。

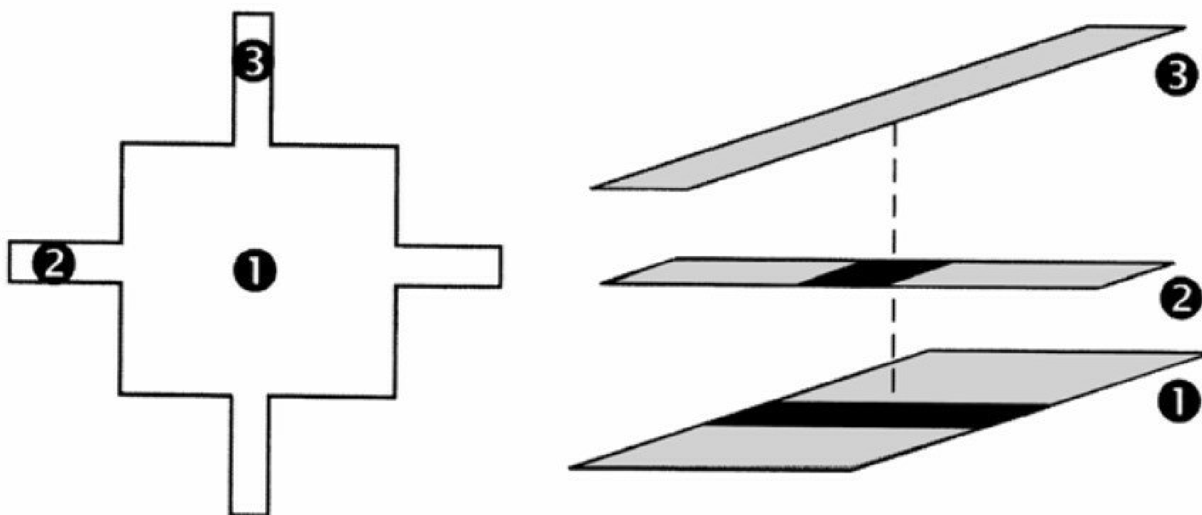


图3-2 带两个沟槽的方切口。左图：成品零件外观；右图：实际上由3个部件组成

……因为部件的顺序必须确定

问题在于，我不能只为这3个部件分别生成数控集命令，然后指望零件就能自动正确地出现——通常特定的顺序是必需的。在本例中，假设要先制作沟槽再制作方切口（如图 3-3 所示），那么在制作方切口时（还记得吗，方切口是用高压冲床冲出来的），金属板材会发生弯曲，因为沟槽将降低金属强度。

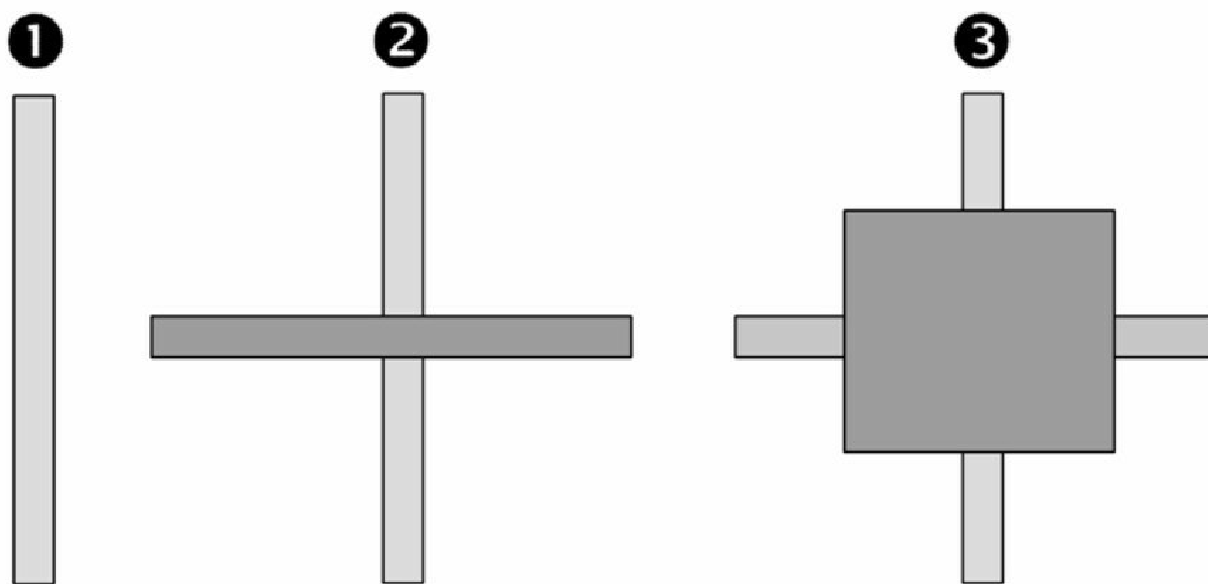


图3-3 糟糕的制作方法。按此顺序，金属板材强度将会降低并且变形

制作图3-2所示的形状时，必须首先冲出方切口，然后再制作沟槽。这是可行的，因为沟槽是用铣床制作的，所使用的是侧向压力。图3-4说明了这一过程。先制作方切口实际上简化了工作而非相反。

幸运的是，专家系统中已经总结出了这些规则，我用不着为此操心了。之所以要花一些时间对这些挑战进行一番解释，是为了使读者了解专家系统所需的信息类型。

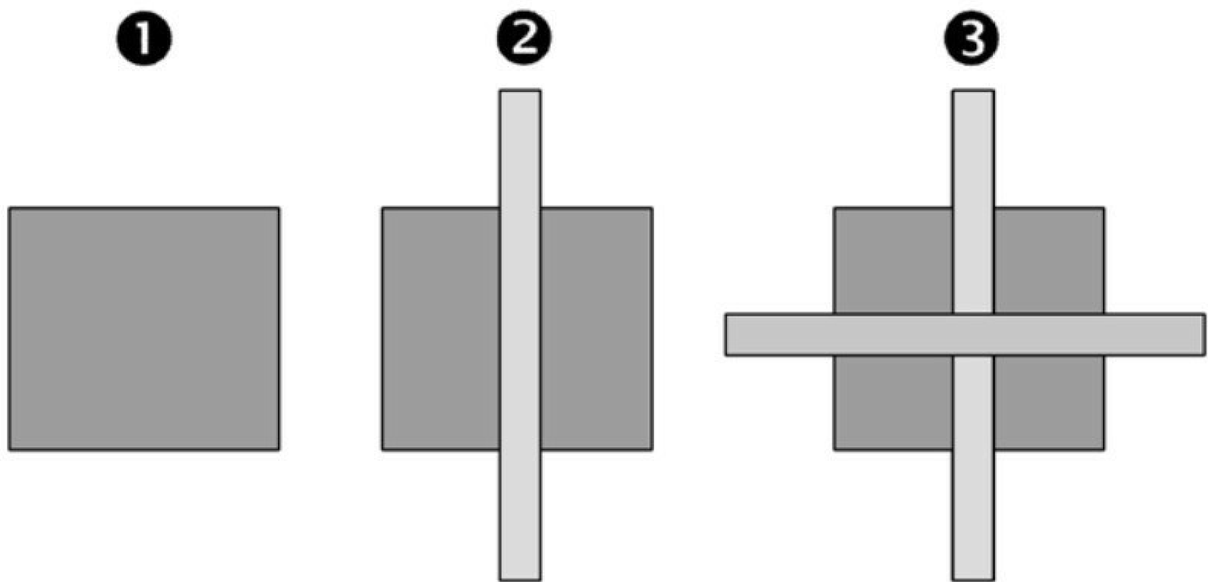


图3-4 制作切口的专家方法。借此能够获得正确的切口

3.5 挑战及其解决方案

挑战：专家系统可以与不断变化的CAD/CAM系统协作

CAD/CAM 系统还处在持续发展和变化之中。我真正要解决的问题是：在CAD/CAM系统变化时，公司仍然能够继续使用昂贵的专家系统。

当时的情况是，公司正在使用的是CAD/CAM系统的一个版本——我们称之为第1版(V1)好了，而新的一个版本——称之为第2版(V2)吧，也将很快推出。虽然这两个版本是由一个厂商开发的，但是彼此并不兼

容。

由于技术和管理上的种种原因，也无法将一个版本中的模型转换到另一版本，因此，专家系统需要支持CAD/CAM系统的两个版本。

除了必须适应CAD/CAM系统的两个不同版本之外，实际上的情况还要更糟。我得知还有一个更新的版本将在不久之后推出，但到底是什么时候还不清楚，而且也不知道它的情况如何。为了保护公司在专家系统上的投资，我希望得到类似于图3-5所示的系统架构。

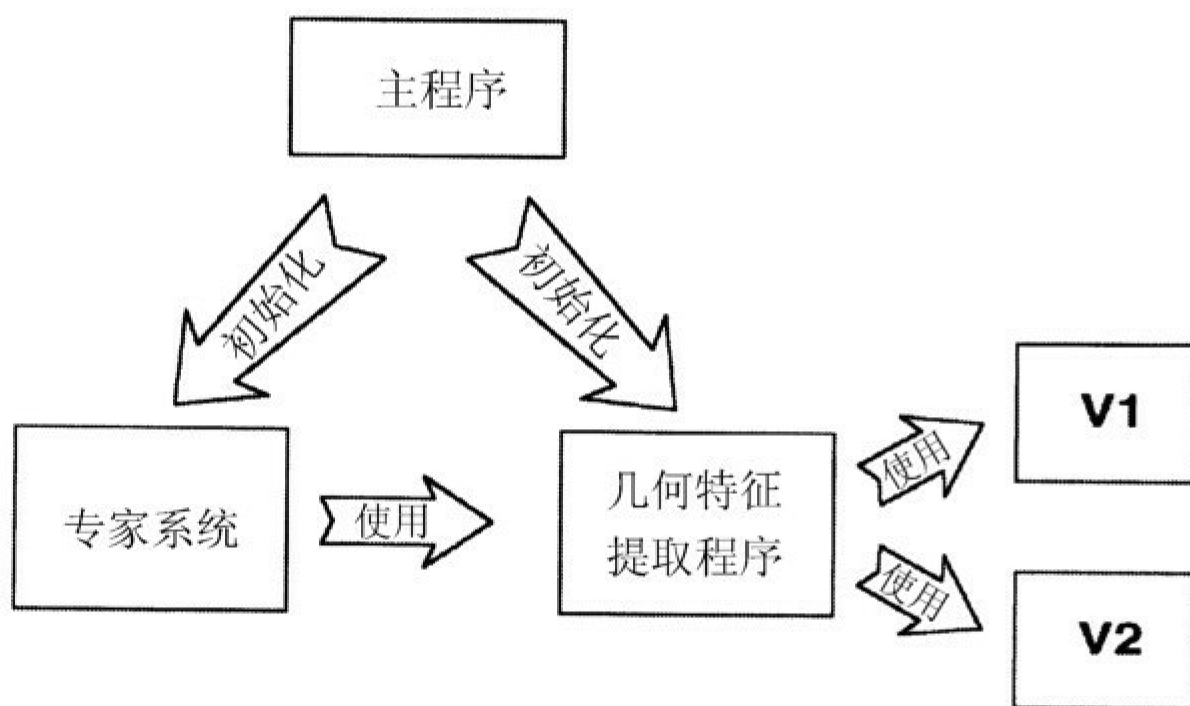


图3-5 解决方案的高层视图

也就是说，应用程序能够初始化一切，从而使专家系统可以使用正确的CAD/CAM系统。但是，专家系统必须两个版本都能够使用。因此，我需要使V1和V2对专家系统而言是相同的。

并不是所有层次都需要多态性

在几何特征提取程序一层上多态性肯定需要，但在部件一层上则不然，这是因为专家系统需要知道所处理的部件类型。无论如何，

CAD/CAM系统的第三个版本推出时，我们不希望还对专家系统进行修改。

高层类图

只要具备对面向对象设计的基本理解就能够看出，系统的高层类图应该类似于图3-6所示。

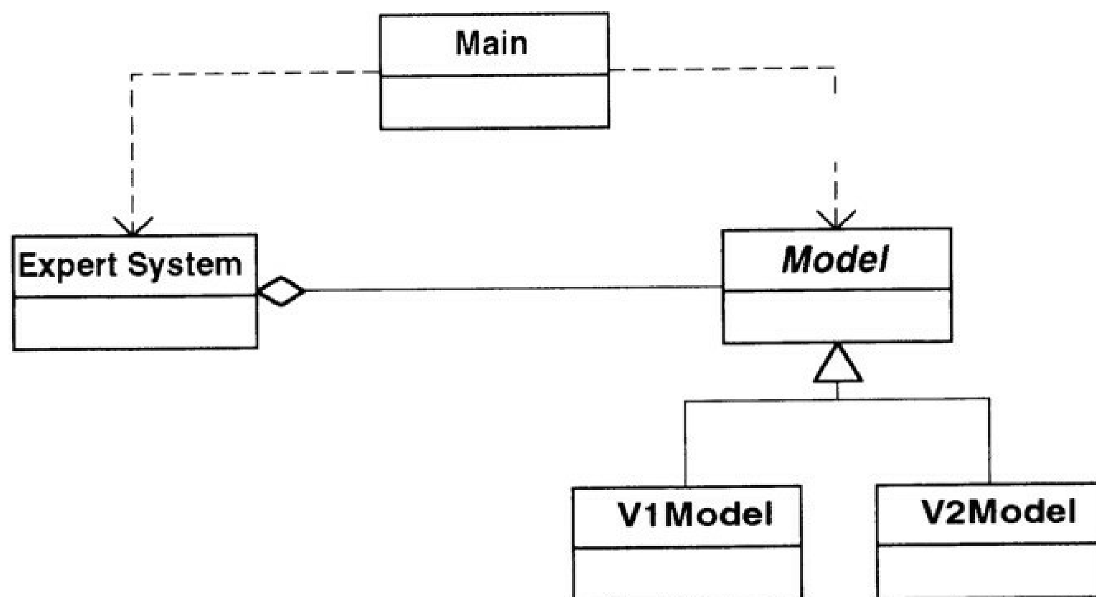


图3-6 解决方案的类图[2]

也就是说，专家系统是通过 Model类与 CAD/CAM 系统发生关系。Main类负责实例化Model类的正确版本（即V1Model或者V2Model）。

CAD/CAM系统更详细的说明

现在来讲讲两个版本的CAD/CAM系统以及它们的工作机理。非常糟糕的是，这两个版本之间的差异很大。

第1版本质上就是一个子例程库的集合。为了获取模型信息，必须进行一系列函数调用。第1版CAD/CAM系统中可能的典型查询过程如下。

1.打开XYZ模型，返回一个指向它的句柄。

2.将该句柄保存为H。

3.对于H所指向的模型，查询其所含部件的数量，保存为N。

4.对于H所指向的模型中的每个部件（1～N）：

a.对于H所指向的模型，查询其第i个元素的标识号，保存为ID。

b.对于H所指向的模型，查询其标识号为ID的部件类型，保存为T。

实例化的正确版本

c.对于H所指向的模型，查询其标识号为ID的部件的X坐标，保存为X。（根据类型，用T确定应该调用哪个例程。）

第1版CAD/CAM系统显然不是面向对象的

这个系统处理起来非常费劲，很显然它并不是面向对象的。无论是谁来使用这个系统，都必须手工维护每次查询的上下文。与部件相关的每次调用，都必须知道部件的类型是什么。

第2版CAD/CAM系统是面向对象的

CAD/CAM 厂商已经认识到这一版本的系统存在固有的局限，构建V2的主要动机就是要使其成为面向对象的。在V2中，几何特征是以对象形式存储的。当系统请求模型时，它将获得代表模型的一个对象。这个模型对象包含一个对象集合，其中每一个都代表一个部件。因为问题领域是以部件为基础的，所以V2用来表示部件的类与前面提到过的部件——沟槽、孔、方切口、特殊形状和不规则形状完全对应，也就不足为奇了。

因此在V2中，可以获得与金属板材的部件相对应的一组对象。这些部件对应的类如图3-7的UML图所示。

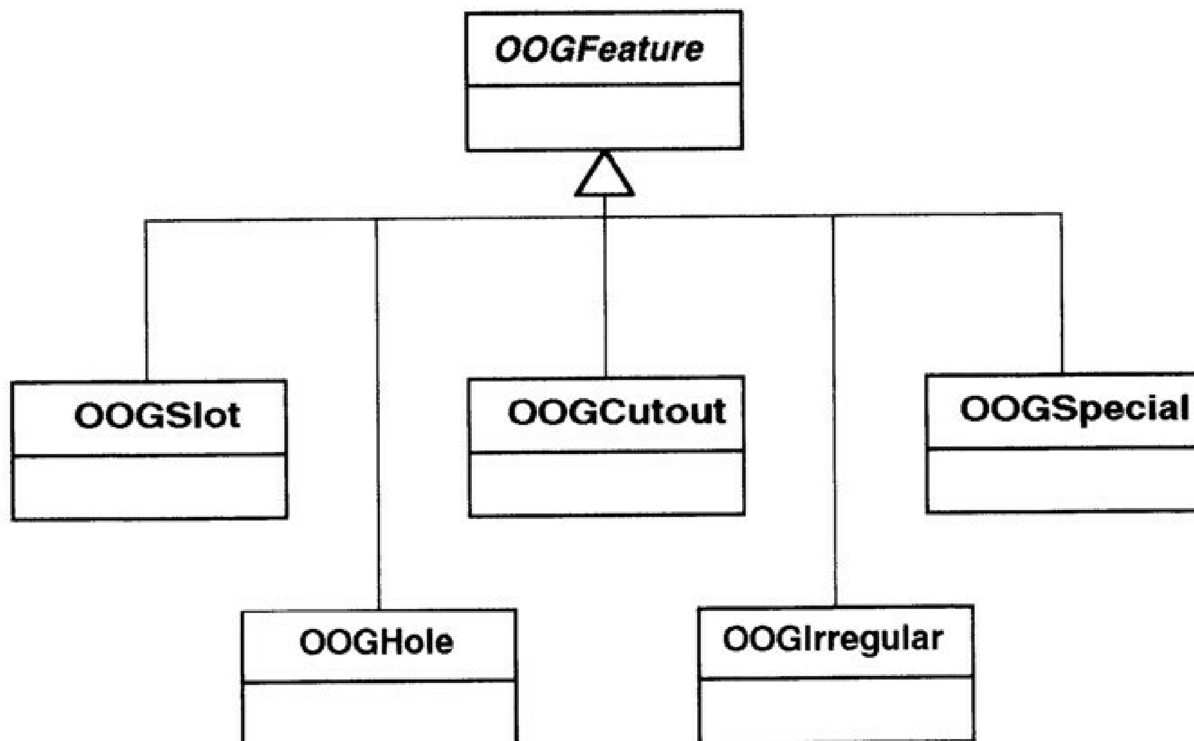


图3-7 V2的部件类

OOG是object-oriented geometry（面向对象几何特征）的简称。这暗示着，V2是一个面向对象的系统。

3.6 小结

本章内容

本章描述了CAD/CAM问题。

必须以相同的方式从不同的CAD/CAM系统中提取信息。这样，公司耗费了巨资的系统（专家系统）就能够继续使用，无需在每次CAD/CAM系统发生变化时进行代价高昂的修改。

两个CAD/CAM系统以完全不同的方式实现，虽然所包含的信息本质上相同。

这个任务与我在许多项目中遇到过的其他问题有许多相似之处：存在具体实现各不相同的多个系统，但是需要让其他对象以相同的方式与

这些不同实现通信。

复习题

简答题

1.本章所描述的系统中要处理的金属板材有哪五个部件？

2.系统的第1版和第2版之间的区别是什么？

阐述题

1.CAD/CAM 问题的核心困难是什么？

2.为什么说多态性在几何特征提取程序一层是必需的，而部件一层则不需要？

观点与应用题

本章中花费了不少时间定义与CAD/CAM 问题有关的术语。

为什么要这样做？

你认为这有必要还是感觉有些离题？

理解用户的术语体系重要吗？

你认为记录用户术语体系最有效的方法是什么？

第4章 标准的面向对象解决方案

4.1 概览

本章内容

本章将为第3章中所讨论的问题提出一个初步的解决方案。这是一次合理的初步尝试，确实能够很快解决问题。但是，它遗漏了一个重要的系统需求：当CAD/CAM系统继续发展变化时应具有的灵活性。

本章中将描述一个基于面向对象的解决方案。这个解决方案并不理想，但可以完成任务。

注意，本章正文中只给出Java代码示例。相应的C++和C#代码示例可以在本书配套网站 http://www.netobjectives.com/dpexplained找到。本书中的所有代码都是这样处理的。

4.2 作为特例来解决

解决方案入手：每个版本都有特定子类

对于第3章中描述的两个不同的CAD/CAM系统，应该如何构建这样一个信息提取系统，使得无论具体CAD/CAM系统是什么样，它对于客户系统都能保持一致？

在思考如何解决这个问题时，我是这样考虑的：如果对沟槽能够解决该问题，那么我就可以将同样的解决方案应用于方切口、孔等等其他形状。接下来考虑沟槽时，我发现能够很容易地特化每一种情况。也就是说，我可以创建一个SlotFeature类，在使用V1系统时从该类派生一个类，在使用V2系统时从该类派生另外一个类。如图4-1所示。

完成解决方案

将这一方法推广到所有部件类型上，解决方案就完成了，如图 4-2所示。

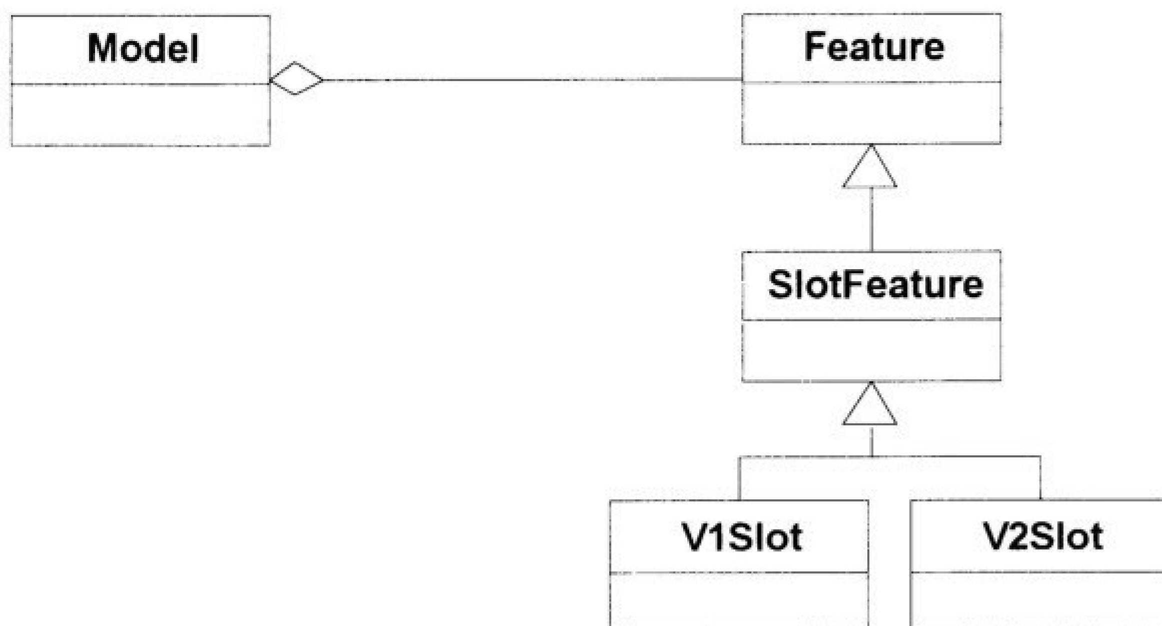


图4-1 沟槽的设计

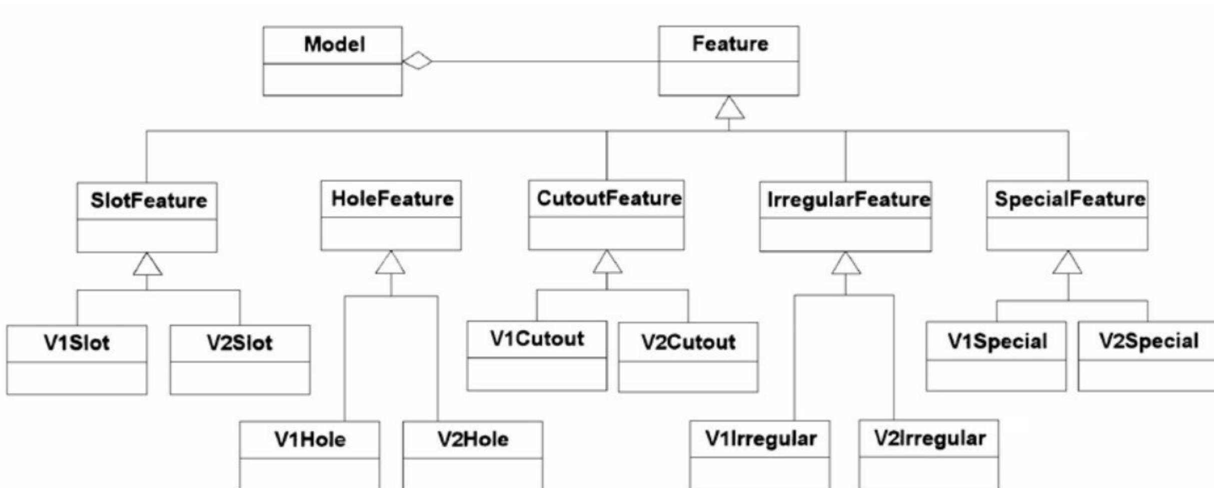


图4-2 信息提取问题的初始解决方案

当然，图4-2还是比较高层的。所有V1xxx类还应该与相应的V1库通信。而所有V2xxx类也应该与V2模型中相应的对象通信。

分别来看每个类，能够更加直观一些：

V1Slot是通过“在实例化时记住自己的所属模型及其在V1系统中的ID”来实现的。然后，只要需要调用V1Slot的某个方法来获取其信息时，该方法就必须调用V1中的一系列子程序，从而获取这些信息。

V2Slot的实现方法与V1Slot很相似，只不过这里每个V2Slot对象应该包含的是V2 系统中与其对应的OOGSlot对象。然后，只要需要请求V2Slot对象的信息时，它将把该请求直接转给OOGSlot对象，再将响应返回发出请求的源客户对象。

图4-3所示为更详细的图，加入了V1和V2系统。

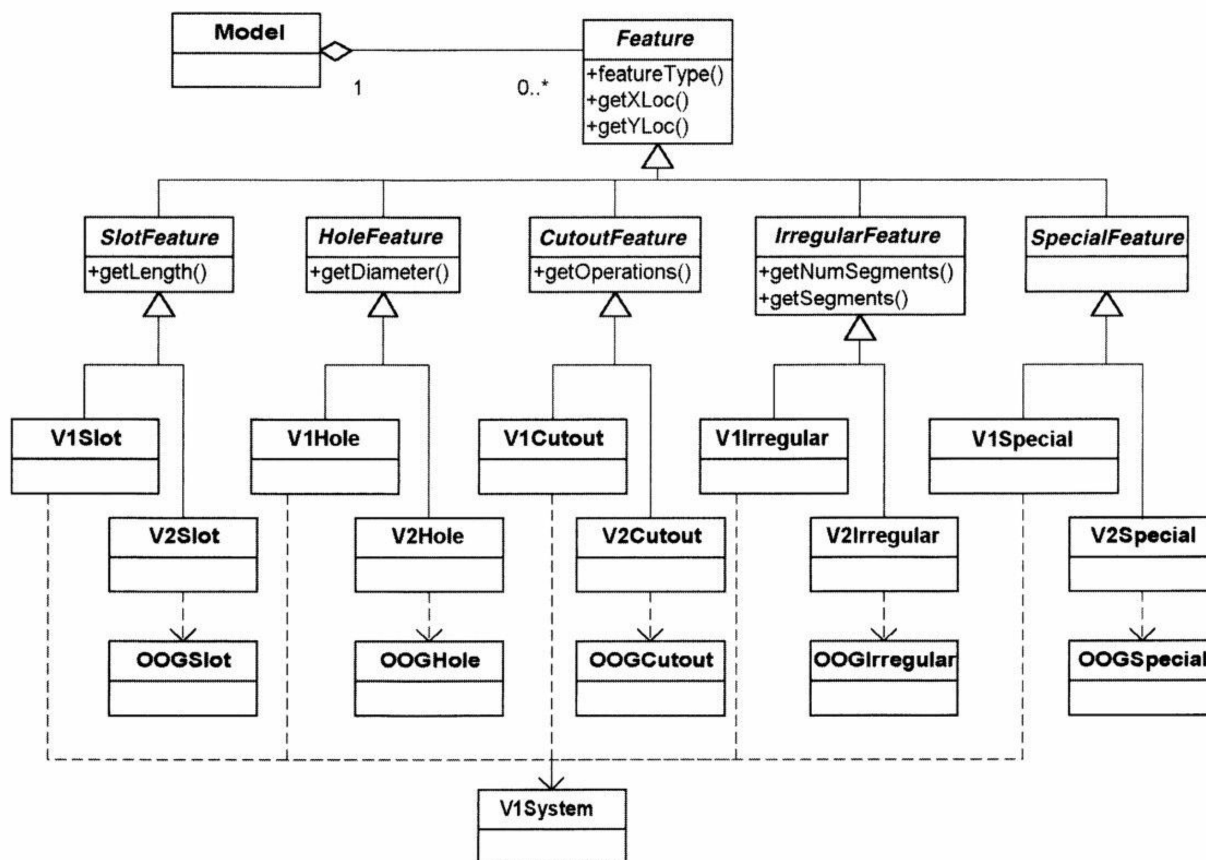


图4-3 初步的解决方案

代码片断有助于理解设计

下面将给出此设计方案中两个类的代码示例。这些示例只是为了帮助你理解如何实现该设计方案。如果你觉得自己能够轻松地实现它，那

就尽管跳过下面的Java示例代码好了。请注意：此处给出的示例是不完整的，只是为了说明问题。要查阅完整的示例代码，请访问本书配套网站：http://www.netobjectives.com/dpexplained。

例**4-1 Java**代码片段：实例化**V1**部件

```
// 实例化部件的代码片段只为演示，不包含错误检查代码
public class V1Model extends Model {

   static public Model buildV1Model (String modelName) {
      V1Model myModel= new V1Model();
      myModel.modelNumber= myModel.openModel( modelName);
      if (myModel.modelNumber <= 0) return null;

      myModel.buildModel();
      return myModel;
```

```
    }

    private void buildModel () {
        // 每个部件都需要知道模型号及自己对应的部件ID以获取信息
        // 注意将这些信息传递给每个对象构造函数的方式

        nElements= getNumberOfElements();
        features= new Feature[ nElements];

        // 针对每一个部件
        int i;
        int ID;
        for (i= 0; i < nElements; i++) {
            // 确定有这个部件创建相应部件对象
            ID= V1.getFeatureID( modelNumber, i);

            switch( V1.getFeatureType( modelNumber, ID)) {
                case FEATURE_SLOT:
                    features[i]=new V1Slot(modelNumber, ID);
                    break;

                case FEATURE_HOLE:
                    features[i]=new V1Hole(modelNumber, ID);
                    break;
                // 其他部件
            }
        }
    }
}
```

例**4-2 Java**代码片段：**V1**方法的实现

```
// myModelNumber和myID是私有成员，
// 包含（V1中）与这个部件对应的模型及部件的信息

public class V1Slot extends SlotFeature {
    . . .
    public double getX1 () {
        return V1.getX1forSlot( myModelNumber, myID);
    }
    public double getX2 () {
        return V1.getX2forSlot( myModelNumber, myID);
    }
}

public class V1Hole extends HoleFeature {
    . . .
    public double getXLoc () {

    return V1.getXforHole( myModelNumber, myID);
   }
}
```

例**4-3 Java**代码片段：实例化**V2**部件

```
// 实例化部件的代码片段只为演示
// 不包含错误检查代码
public class V2Model extends Model {

    static public Model buildV2Model (String modelName) {
        // 打开模型
        V2Model myModel= new V2Model();
        if (!myModel.openModel( modelName)) return null;

        myModel.buildModel();
        return myModel;
    }

    private void buildModel () {
         // 每个部件对象都需要知道自己在V2系统中对应的部件以获取信息
         // 注意将这些信息传递给每个对象构造函数的方式

        nElements= getNumberOfElements();
        OOGFeature oogF;

        // 针对每一个部件
        int i;
        for (i= 0; i < nElements; i++) {
            // 确定有这个部件创建相应部件对象
            oogF= getElement(i);
            switch( oogF.getType()) {
                case OOG_SLOT:
                    features[i]= new V2Slot( oogF);
                    break;

                case OOG_HOLE:
                    features[i]= new V2Hole( oogF);
                    break;
                // 其他部件
            }
        }
    }
}
```

例**4-4 Java**代码片段：**V2**方法的实现

// oogF是对V2中对应的部件对象的引用

```
public class V2Slot extends SlotFeature {
    . . .
    public double getX1 () throws Exception {
        // 调用oogF上相应的方法以取得所需信息
        return myOogF.getX1Loc();
    }
    public double getX2 () throws Exception {
        // 调用oogF上相应的方法以取得所需信息
        return myOogF.getX2Loc();
    }
}
public class V2Hole extends HoleFeature {
    . . .
    public double getXLoc () throws Exception {
        // 调用oogF上相应的方法以取得所需信息
        return myOogF.getXLoc();
    }
}
```

解决方案满足了一个目标：通用API

在图4-3中，我添加了部件需要的一些方法。请注意它们因部件的类型不同而各不相同，这意味着我不可能获得跨部件的多态。但是这不是什么问题，毕竟专家系统需要知道自己具有哪些部件类型。之所以这样说，是因为对不同类型部件，专家系统需要获得不同的信息，从而会根据给定的零件上有哪些部件，进一步做出决策。

因此，我无需考虑部件的多态。相反，我需要的是不修改专家系统，就能“即插即用”不同的CAD/CAM系统。

死死盯住球

打垒球的一个基本原则，就是死死盯住球。不要因为次要的细节而分散注意力。将精力集中在手头最重要的任务上，现在要处理的事情上。

对于软件开发人员来说，这同样是金玉良言。正确的分析和设计要求我们在互相矛盾的关注点之间找到平衡。必须决定问题的哪些方面是设计的重点，或者应该让系统防范哪些变化。寻找平衡必须成为做出设计决策的第一要务。不要让其他细节转移了你的注意力。

在本例中，我们已经预见到未来 CAD/CAM 程序会发生变化。而这种变化将产生重大影响，因此我们必须予以控制。这就是我们要盯住的“球”。

还存在很多问题

我现在要解决的事情——透明地处理多个CAD/CAM版本，提供了几条线索，暗示我这个解决方案并非上策。

方法之间存在冗余——可以很容易地想象到调用V1系统的那些方法存在许多相似之处。例如，Slot的V1getX方法和Hole的V1getX方法将非常相似。

杂乱——尽管这一点并不总是意味着存在问题（而且杂乱与否主观性很强），但这一因素使我进一步对这个解决方案产生了怀疑。

紧耦合——这个解决方案存在紧耦合，因为这些部件是间接地相互关联着的。这些关联本身暗示，如果出现以下情况，可能需要修改所有部件：

必须改用一个新的CAD/CAM系统；

修改了现有的CAD/CAM系统。

弱内聚——内聚性非常弱，因为执行核心功能的方法分散在多个类中。

毕竟，我最关心的是要为未来做准备。想象一下吧，一旦要改用CAD/CAM系统的第三个版本，情况会怎样？各种变化的灾难性组合会

置我们于死地的！请看图4-3中第三排类图。

其中有5种部件。

每种部件有一对类，每个CAD/CAM系统一个。

如果有了CAD/CAM系统的第三个版本，每种部件将有3个类。

因此我将有15个类而不是10个类。

这肯定不是一个我愿意维护的系统！而且再想象一下，如果其他的东西也发生变化，情况又会怎样？现在，我只需处理两方面的变化（部件的类型和数据存在哪个系统，V1 还是 V2），因此所拥有的类的数目是部件数乘以系统数。如果还要处理另一方面的变化，这种方法很快就会完全失去控制。

分析陷阱：过早过多地关注细节

分析人员都可能犯的一个常见错误是：在开发过程中过早地深入细节。这很自然，因为处理细节比较容易。细节的解决方案总是显而易见，但并非肯定是最好的入手点。细节问题的处理应该尽可能推后。

在本例中，我已经实现了一个目标：为部件信息提供了一个公用的API。我还从责任的角度定义了对象。但是，这样做的代价是所有东西都要作为特例。如果有了新的特例，我还要如法炮制地将它实现一遍。因此，维护代价太高了。

……直觉告诉我，肯定有更好的解决方案

这是我最初的拙作，我很快就对它产生了不满意的感觉。这种感觉更多来自直觉，而不是前面所提到的更合乎逻辑的原因。我感到设计有问题。

对这个问题，我强烈地感到存在更好的解决方案。但是，两小时过去了，我仍然提不出更好的方案。问题似乎出在我所用的方法上，本书后面将对此继续进行讨论。

留意你的直觉

令人惊讶的是，本能直觉能够很好地评判设计质量。我建议开发人员应该学会倾听自己的心声。

这里的“本能直觉”，是指看到某些不喜欢的东西时身体的第一感觉。我知道这听上去好像不太科学（实际上确实如此），但我的经验不断证明：当我直觉上不喜欢某个设计时，更好的设计肯定还隐藏在某个角落里。当然，有时附近会有几个不同的角落，我并不是总能确定哪个角落里会隐藏着更好的设计。

4.3 小结

本章内容

本章说明，将一切都作为特例来解决问题是非常容易的。这种解决方案直截了当。我可以借此在不改变已有系统的情况下添加新的方法。但是，这种方案有几个缺点：高冗余、低内聚和（未来发生变化时会出现的）类爆炸。

过分依赖继承将带来超出正常的维护成本（至少，我感觉会是这样）。

复习题

简答题

1.找出图4-3中UML图的以下元素：

抽象类

重数

派生

组合方法

公开方法

2.CAD/CAM应用程序所必需的基本能力是什么？

3.初步的解决方案存在四个问题，它们分别是什么？

阐述题

描述解决CAD/CAM问题的初步解决方案。这是合理的初步解决方案吗？

观点与应用题

1.尽可能晚地深入细节。对此你同意吗？为什么？

2.一个解决方案被放弃了，因为直觉告诉我这不是一个好的解决方案。对于分析员/程序员，这样凭直觉做事合适吗？

---

[1].CAD：Computer Aided Drafting，计算机辅助设计；CAM：Computer Aided Manufacturing，计算机辅助制造。

[2].此图和本书中的其他类图都使用UML表示法。请参阅第2章中对UML表示法的描述。

第三部分 设计模式

概览

本部分内容

本部分将介绍设计模式：什么是设计模式，如何使用设计模式。将讲述与CAD/CAM问题（见第3章）有关的几个模式。我将采取这样的方式，先分别介绍各个模式，然后讲解如何将它们与前述问题联系起来。在学习模式的所有章节中，我将始终强调GoF（“四人帮”人们对《设计模式》作者Gamma、Helm、Johnson和Vlissides的昵称）在其开创性的著作《设计模式：可复用面向对象软件的基础》中所提倡的面向对象策略。

章 讨论的主题

5 设计模式简介

设计模式的概念、它们的建筑学起源以及在软件设计领域中如何应用设计模式。

学习设计模式的动机。

6 Facade模式

是什么，用在何处和怎样实现。

如何将Facade模式与CAD/CAM问题联系起来。

7 Adapter模式

是什么，用在何处和怎样实现。

Adapter模式与Facade模式的比较。

如何将Adapter模式与CAD/CAM问题联系起来。

8 开拓视野

面向对象程序设计中的一些重要概念：多态、抽象、类和封装。

本章将用到第5章到第7章中学到的知识。

9 Strategy模式

第一次将模式当作一种处理问题领域中变化的方式来讲述。

10 Bridge模式

这个模式比前面的模式要复杂得多。不过它的用处也大得多，所以，我将更详细地对它进行探讨。

如何将Bridge模式与CAD/CAM问题联系起来。

11 Abstract Factory模式

这个模式主要用来创建一组对象。

是什么，用在何处和怎样实现。

如何将Abstract Factory模式与CAD/CAM问题联系起来。

目标

学完本部分，你将理解设计模式是什么，知道它们为什么有用，并且熟悉几个具体的模式。你还能看到如何将这些设计模式与前面的CAD/CAM问题联系起来。但是，光凭这些信息，还不足以创建出比前面过分依赖继承所得到的解决方案更好的设计。然而，这将为我们以更佳的方式使用模式打下基础，而不仅仅是重视其解决方案，这种方式与许多设计模式使用者的大不相同。

第5章 设计模式简介

5.1 概览

本章内容

本章将介绍设计模式的概念。

讨论设计模式的建筑学起源，以及在软件设计领域中如何应用设计模式。

考察学习设计模式的动机。

设计模式和面向对象设计是相得益彰的关系

设计模式是面向对象技术的最新进展之一。现在面向对象分析工具、图书和培训都在加入设计模式的内容，设计模式学习小组在各地的发展如火如荼。通常的建议，都是在掌握了基本面向对象技术之后，再学习设计模式。但我发现事实恰恰相反：在学习面向对象技术过程中较早地学习设计模式，对于加深面向对象分析与设计的理解大有裨益。

在本书后面的内容中，我将不仅讨论设计模式，还将讨论它们怎样展示和加强优秀的面向对象原则。我希望这些内容能增进读者对这些原则的理解，说明为什么所讨论的设计模式代表了优秀的设计。

别着急

这些内容有些可能看上去有些抽象或者哲学化。不要着急！本章将为你理解设计模式奠定基础。理解这些内容，将提高你理解和使用新模式的能力。

我的许多思想都来自Christopher Alexander的The Timeless Way of Building[1]一书。对这些思想的讨论将贯穿本书始终。

5.2 设计模式源自建筑学和人类学

多年以前，有一位名叫Christopher Alexander的建筑师这样扪心自问："质量能够客观评价吗？"也就是说，真地会情人眼里出西施吗？人们是否能够就哪些东西美哪些东西不美取得共识？Alexander 感兴趣的特殊形式的美是一种建筑质量：什么能让我们领会到一个建筑设计是否优秀？假设一个人要为一座房子设计入口，他怎样才能知道设计是否优秀呢？我们能够评价设计是否优秀吗？这种评价有客观根据吗？有能够描述我们共识的根据吗？

质量能够客观评价吗？

Alexander 认为建筑系统中存在这样的客观根据。评价建筑美观与否不只是个人喜好，我们能够通过可以度量的客观根据描述美。

文化人类学中也出现了类似的观点。该学科的大量工作说明，在一种文化中，不同的个人在很大程度上会就"何为优秀的设计，何为美"之类的问题达成共识。文化对优秀设计的评价将超越个人的看法。我相信这种超越性的模式能够作为评价设计的客观基础。文化人类学的一个重要分支，就是寻找描述一种文化的行为和价值观的模式。[2]

设计模式背后的一个观点，就是软件系统的质量可以客观度量。

怎样才能可重复地获得优秀设计？

如果你认同这样的观点，质量优秀的设计是存在共识，可以进行描述的，那么应该怎样着手进行优秀的设计呢？我可以想象到Alexander这样扪心自问：

在优秀设计中具备而在劣质设计中不具备的是什么？以及

在劣质设计中具备而优秀设计中不具备的又是什么？

这些问题来源于Alexander的如下信念，如果设计的质量可以客观评价，我们就应该能够可以找出是设计因何优秀，又因何拙劣。

寻找共性

为研究这一问题，Alexander 对建筑物、城镇、街道等等实际上人类为自身所建造的各种生活空间的方方面面进行了大量观察。他发现，在特定的建筑物中，优秀的结构都有一些共同之处。

……特别是要解决的问题特征中的共性

各种建筑结构即使属于一种类型，也会互不相同。虽然它们互不相同，但可能都具有很高的质量。

例如，两个门廊结构上也许不同，但是，仍然可能都具有高质量。它们可能是要为不同的建筑解决不同的问题。一个门廊可能是作为走道和前门之间的过渡，而另一个门廊可能是为了在天气炎热时提供阴凉。或许，两个门廊在解决同一个问题（过渡）时，也可能采用不同的方式。

Alexander 看到了这一点。他知道结构不能与要解决的问题分离，因此，在寻找和描述设计质量一致性的探求中，Alexander 认识到，必须观察为了解决同样的问题而设计的不同结构。例如，图5-1中显示了对“区分入口”这一问题的两种解决方案。

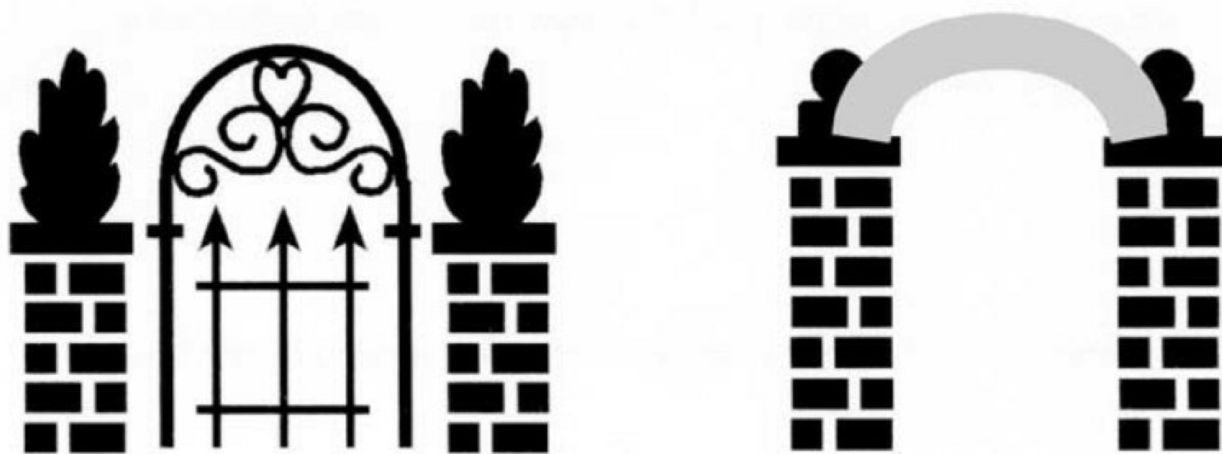


图5-1 两个结构看似不同，但解决的是同一个问题

引出模式的概念

Alexander发现，通过这样的方式——观察解决相似问题的不同结构，可以缩小关注范围，从而看清优秀设计之间的相似之处。他将这种

相似之处称为模式。

他对模式的定义是“在某一背景下某个问题的一种解决方案”：

每个模式都描述了一个在我们的环境中会不断重复出现的问题，并进而叙述了这个问题解决方案的要素，通过这种方式，解决方案能够百万次地反复应用，但是具体方式又不会完全相同。[3]

我们来回顾一下A1exander对此的阐释文字。表 5-1 中从他的The Timeless Way of Building[4]一书中摘录了一些段落。这本书简洁地说明了模式的基本原理。

表5-1 The Timeless Way of Building的片段

Alexander 如是说	我的评述……
同样，一个处理得很好的庭院有助于使人们在里面充满生气	模式都有一个名称和一个目的。此处，模式的名称是“庭院”，目的是“使人活跃”、“充满生气”
设想一下庭院中起作用的各种因素。其中最基本的是，人们需要某种私密的室外空间，在那里他们可坐在蓝天之下，仰观星群，沐浴阳光，可能还要种植花木。这是显而易见的	虽然有时候要解决的问题可能显而易见，但明确地说明它是非常重要的，这个问题是提出模式的第一条理由。Alexander 在此对“庭院”模式正是这样做的
但是还有更微妙的因素。例如，倘若庭院太封闭，看不到外界，人们就会感觉不舒服，想要出去……他们需要看到外面更大、更远的空间	他指出了一个用简化的解决方案难以解决的问题，然后提供了一种解决该问题的方法
而且，人都有自己的习惯。如果他正常地生活，每日都在这个庭院中进进出出，庭院就会变得可亲，变成一个很自然的去处……庭院于是适得其所	有时我们会因为太熟悉而看不到一些显而易见的事情。模式的价值就在于，那些经验不多的人也可以利用前人获得的经验：既包括实现优秀的设计必不可少的，也包括避免拙劣的设计必须远离的
但是，如果庭院只有一条通路，一个你“想到”时才去的地方，它是不会使人感觉可亲的，很可能门可罗雀……人们会更多地去那些亲近的去处	虽然模式经常被认为是理论上的概念，但是事实上它们所反映的是过去不断出现的实际问题
又如，突然从庭院内直接迈到院外，肯定会有不连贯的感觉……这一点很微妙，但足以使人不快。	Alexander 描述了一种事实上存在（而且重要），但是很容易被忽视的因素
如果有转换空间——一个有顶但却开敞的外廊或走廊，那么它就是一个室内、室外间心理上的过渡，能够更容易、更简单地逐渐将你带入庭院……	对于建造舒适庭院存在一个可能被忽视的挑战，他为此提出了一种解决方案，而且他还说明一个模式（这里的“庭院”）经常能够为另一个模式（这里的“外廊”或者“走廊”）提供背景。模式之所以能够互相提供背景，是因为发现模式有助于澄清问题所在，从而使其他模式更容易显现出来
当在庭院中能够外眺更大的空间，有来自各个房间的通路，有外廊或走廊时，这些因素可以自行疏解。外眺的景致使其舒适，多一条通路有助于产生一种习惯感觉，走廊使人们更容易更经常地进入……庭院逐渐成为一个舒适习惯的场所	A1exander 在告诉我们如何建造一个舒适的庭院……以及它为何舒适

每个模式描述必须有4个部分

总结一下，Alexander说一个模式的描述应该包括4项：

模式的名称；

模式的目的，即要解决的问题；

实现方法；

为了实现该模式我们必须考虑的限制和约束因素。

几乎任何设计问题中都存在模式

A1exander 认为，模式可以解决可能遇到的几乎所有建筑问题。他还进而认为模式可以结合起来解决更复杂的建筑问题。

……而且可以结合起来解决复杂问题

关于多个模式的结合，本书将在后面讨论。现在我想主要讲一讲他谈到的模式如何解决特定问题。

5.3 从建筑模式到软件设计模式

这些建筑学的知识和我们软件开发人员又有什么关系呢？

将Alexander的思想用于软件

20世纪90年代初，一些聪明的开发人员偶然接触到Alexander有关模式的工作。他们很想知道，在建筑学成立的理论，是否在软件设计中也适用[5]。

软件中是否存在不断重复出现、可以以某种相同方式解决的问题？

是否可能用模式方法来设计软件，即先找出模式，然后根据这些模式创建特定的解决方案？

这些开发人员感到，这两个问题的答案都毋庸置疑是肯定的。接下来要做的，就是找出一些模式，然后制定出新模式的编录标准。

GoF在设计模式方面的早期工作影响深远

虽然在20世纪90年代初许多人都在研究设计模式，但对这个羽翼未丰的社区影响最大的书却是由Gamma、Helm、Johnson和Vlissides合著的《设计模式：可复用面向对象软件的基础》[6]。为了褒奖他们的重要

工作，大家给四位作者取了一个通行的昵称——GoF。

这本书有以下几个目的。

将设计模式的思想应用于软件设计——并称它们为设计模式（design pattern）。

给出了编录和描述设计模式的一种格式。

编录了23个设计模式。

在这些设计模式的基础上推导出了一些面向对象的策略和方法。

GoF自己并没有创造书中的模式，认识到这一点很重要。相反，他们只是将软件界已经存在的、反映了（针对各种具体问题的）优秀设计经验的模式识别出来。（请注意，这与Alexander的工作是类似的。）

今天，已经有好几种不同格式来描述设计模式。因为本书并不是要讲述如何编写设计模式，所以我不会评价哪一种描述模式的格式最好，但是，任何描述中都需要包含表5-2中所列的项目。

表5-2 模式的关键特征

项　目	描　述
名称	每个模式都有唯一的用于标识的名称
意图	模式的目的
问题	模式要解决的问题 ①
解决方案	模式怎样为问题提供适合其所处环境的一个解决方案
参与者和协作者 ②	模式所涉及的实体
效果	使用模式的效果，研究模式中起作用的各种因素
实现	模式的实现方式 注意：实现只是模式的具体体现，而不能视为模式本身
一般性结构	显示模式典型结构的标准图

①《设计模式》一书中称为“动机”，包括问题及其所处环境。——译者注

② 指参与模式的类和/或对象。——译者注

本书中讨论的所有模式，我都将用一页左右的篇幅根据模式的这些关键特征进行总结。

后果/约束

设计模式中所用的术语后果（consequence）常常被人误解。在英文的日常用法中，该词总是暗含贬义（你永远不会听到有人说："我的彩票中奖了！其后果是，我现在用不着再去工作了！"）。然而，在设计模式界，这个词只是指因果中的效果而已。也就是说，如果你用如此如此这般方法实现了这个模式，它将怎样影响已有因素，又会怎样被已有因素所影响。

注意：除非特别提到，本书中的关键特征总结都摘自《设计模式》一书。

5.4 为什么学习设计模式

设计模式有助于复用和沟通

现在你对"什么是设计模式"已经有了感性认识，也许有人会问："为什么要学习设计模式呢？"原因有很多，一些非常明显，而另一些则不那么明显。

学习模式最常见的理由是因为我们可以借其：

复用解决方案——通过复用已经公认的设计，我能够在解决问题时取得先发优势，而且避免重蹈前人覆辙。我可以从学习他人的经验中获益，用不着为那些总是会重复出现的问题再次设计解决方案了。

确立通用术语——开发中的交流和协作都需要共同的词汇基础和对问题的共识。设计模式在项目的分析和设计阶段提供了共同的基准点。

设计模式提供了观察分析和设计的更高视角

模式还为我们提供了观察问题、设计过程和面向对象的更高层次的视角，这将使我们从"过早处理细节"的桎梏中解放出来。

等你读完本书的时候，我希望你将同意这是学习设计模式的最重要的原因之一。它将改变你的思维定式，使你成为更加高效的分析人员。

“细节桎梏”实例：木匠制作一组抽屉

为了说明这一优点，我想引述一段两个木匠之间关于“如何为橱柜制作抽屉”的谈话。[7]

想象一下，有两个木匠在讨论怎样为橱柜制作抽屉。

木匠甲：你认为我们应该怎样制作这些抽屉？

木匠乙：这个嘛，我想榫子应该这样做：在木料上直着锯下去，然后向回转45°再锯；接着再直着锯，然后换一个方向45°往回锯；接着再直着锯下去，然后……

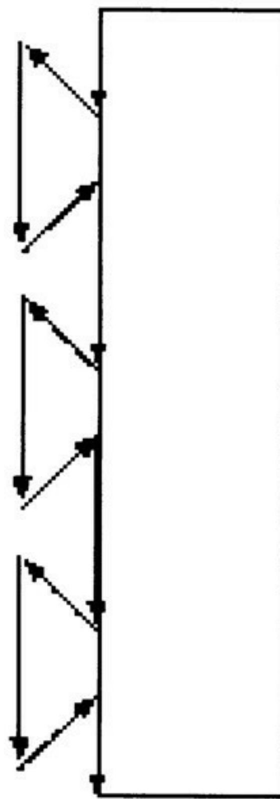
细节可能使解决方案混乱不明

现在，你要做的就是搞清楚他们说的是什么意思！

这段描述是不是让人不知所云？木匠乙到底给出了什么建议？细节往往就是如此！让我们试着将他的叙述画出来。

木匠说……	画出来就是……
“我想榫子应该这样做：在木料上直着向下锯，然后回转 45° 再锯……”	
“……接着再直着锯，然后转 45° 往回锯，再直着锯下去，然后……”	

（续）

木匠说……	画出来就是……
“……最后就做出了一个鸠尾榫。我想应该是这样的！”	

这听起来多么像代码评审：细节，细节，还是细节

这听上去像不像似曾相识的代码评审？在评审中有一位程序员这样描述自己的代码：

然后，我在这里用一个 WHILE 循环来……接着是一系列 IF 语句执行……这里我用一条SWITCH语句处理……

你获得的是对代码细节的描述，而对“程序到底要做什么”、“为什么这么做”，你却毫无头绪!

木匠可不会真地讨论得这么细节

当然，正经的职业木匠可不会这样说话。真实的情形应该是这样：

木匠甲：我们应该用鸠尾榫还是斜榫？

看到这里的本质区别没有？木匠们现在讨论的是一个问题的解决方案上的本质差异，他们的讨论层次更高、也更抽象了，从而避免了陷入具体解决方案的细节泥沼中。

当木匠谈到“斜榫”时，他的脑子里已经对这个解决方案浮现出如下特征：

它是一个更简单的解决方案——斜榫更容易制作。只需将制作榫的木料锯出45°斜面，然后用钉子或者木胶接合起来即可（如图5-2所示）。

它更轻型——斜榫比鸠尾榫强度低。在重压下，将无法保持榫接。

它不太引人注目——斜榫的一个锯面，与鸠尾榫的多个锯面相比，更不显眼。

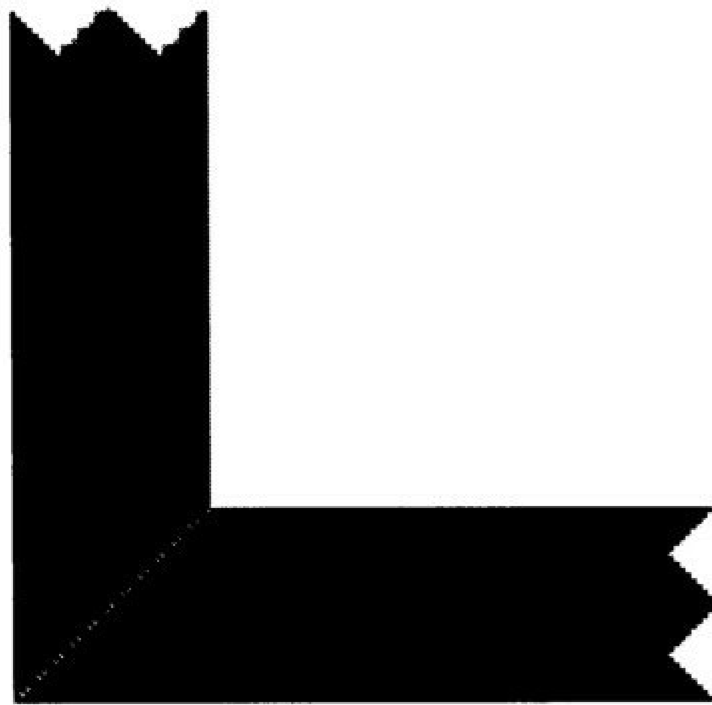


图5-2 斜榫

当木匠谈到“鸠尾榫”时，他的脑子里浮现出另一些特征。这些特征对外行来说可能并不明显，但任何一位木匠都会明白如故。

它是一个更复杂的解决方案——制作鸠尾榫涉及的问题更多。因此，它的成本也更高。

它不容易受温度和湿度影响——当温度和湿度变化时，木材会膨胀或收缩，但是，鸠尾榫仍然能够保持坚固。

它与紧固系统无关——事实上，鸠尾榫甚至不需要依赖胶水。

它看上去更赏心悦目——如果制作精良，会很美观。

也就是说，鸠尾榫是一个坚固、可靠、美观的榫，但制作复杂（所以成本也比较高）。

高层次的对话在继续

所以，当木匠甲这样问的时候：

我们应该用鸠尾榫还是斜榫？

他真正要问的问题是：

我们是应该用一个制作昂贵但美观耐用的榫，还是应该只用一个制作快速而且不美观的榫，能坚持到检查结束就行？

我们应该说，木匠们的讨论其实是在两个层次上进行的：他们话语表面上的层次，和谈话真正的内容，层次更高，外行听不出来，而其中含义却非常丰富。这种更高的层次就是“木匠模式”的层次，它反映了木匠眼中的真正的设计问题。

在第1种情形中，木匠乙讨论的是榫的实现细节，反而使真正的问题模糊不清。在第2种情形中，木匠甲要根据榫的成本和接合性质来决定使用哪种榫。

谁更有效率呢？你更愿意与谁一起工作？

模式能够帮助我们既见树木，又见森林

当我说“模式有助于提高思考层次”时，其中就蕴涵着这一层含义。从本书后面的内容中你将了解到，如果能够这样提高自己的思考层次，新的设计方法也将浮现出来。这正是模式真正的威力所在。

5.5 学习设计模式的其他好处

改善团队的沟通和个人学习

我自己在开发团队中使用设计模式的经验证明，设计模式既可以帮

助开发人员个人的学习，也可以帮助团队提高。这是因为，经验较少的团队成员能够亲眼看到已经掌握设计模式的资深开发人员如何从中获益，他们会更加自发、主动地学习这些强大的知识。

代码更易于修改和维护

大多数设计模式还能使软件更容易修改和维护。其原因在于，它们都是久经考验的解决方案。所以，它们的结构都是经过长期发展形成的，比新构思的解决方案更善于应对变化。而且，这些模式所用代码往往更易于理解——从而使代码更易维护。

设计模式阐述了基本的面向对象原则

如果正确教授，设计模式能够大大加深对基本面向对象设计原则的理解。我已经在教授面向对象入门课程中无数次见证了这一点。在课程中，我首先对面向对象范型进行简短介绍，然后就开始教设计模式，用它们来阐述基本的面向对象概念（封装、继承和多态）。在为期3天的课程结束后，尽管我们大部分时间里讨论的是模式，但是这些基本概念——许多学员都是刚刚接触——似乎都已经是老朋友了。

采用更好的策略，即使不用模式

《设计模式》一书对优秀面向对象设计的策略提出了一些建议。其中包括以下几点。

按接口编程（designing to interface）。

尽量用聚合代替继承。[8]

找出变化并封装之。

这些策略在本书讨论的大多数设计模式中都用到了。用不着学习太多设计模式，只学几个就能使你理解这些策略的重要性。这种理解将会成为将策略应用于实际设计问题的一种能力，即使你并不直接使用设计模式。

学会巨型继承层次结构的替代方案

设计模式还有一个好处是，你或你的团队可以在不使用巨型继承层

次结构的情况下，为复杂问题创建出设计方案。同样，即使并不直接使用设计模式，不使用巨型继承层次结构也会使设计质量提高。

5.6 小结

本章内容

本章讲述了“什么是设计模式”。Christopher Alexander 说：“模式是在某一背景下某个问题的一种解决方案。”它们绝不只是解决某人个别问题的模板。它们是描述动机的一种方式，不仅包括我们要得到的效果，也包括困扰我们的问题。

本章探讨了学习设计模式的理由。学习模式有助于：

对不断重复出现问题，复用既有的、高质量的解决方案；

确立通用的术语，改善团队内的沟通；

提升思考层次；

判断设计是否正确，而不仅仅是能够奏效；

改善个人学习和团队学习；

提高代码的可修改性和可维护性；

采用更佳设计方案，即使没有明确使用模式；

发现巨型继承层次结构的替代方案。

复习题

简答题

1.设计模式思想应该归功于谁？

2.Alexander发现，通过观察解决类似问题的结构，能够看清什么问题？

3.给出模式的定义。

4.设计模式的描述中关键要素是什么？

5.学习设计模式的三个原因是什么？

6.《设计模式》一书对优秀面向对象设计的策略提出了哪些建议？

阐述题

1.“有时我们会因为太熟悉而看不到一些显而易见的事情。”模式在哪些方面能够帮助避免这种现象？

2.《设计模式》一书中编录了23个模式。这些模式来自哪里？

3.模式中“因素”和“效果”的关系是什么？

4.你认为“找出变化并封装之”是什么意思？

5.为什么应该避免巨型继承层次结构？

观点与应用题

1.举出一个令人感觉“死气沉沉”的建筑或者结构。它不具备看上去更加“生机勃勃”的类似结构哪些共有特质？

2.“模式有助于提高思考层次。”你有过什么类似经历吗？举出一个例子。

第6章 Facade模式

6.1 概览

本章内容

我们将从 Facade 模式开始学习设计模式。这个模式你很可能已经实现过，只是不知道它的名字罢了。

在本章中，我们将：

解释Facade模式是什么，用在何处；

给出Facade模式的关键特征；

给出Facade模式的一些变体；

将Facade模式与CAD/CAM问题联系起来。

6.2 Facade模式简介

意图：一个一致的高层接口

《设计模式》一书中对Facade模式的意图是这样叙述的：

为子系统中的一组接口提供一个统一接口。Facade模式定义了一个更高层的接口，使子系统更加容易使用。[9]

这段话大致是在说：我们需要用一种比原有的方式更简单的办法与系统交互，或者说，我们需要以一种特殊的方式使用系统（例如以二维的方式使用一个三维绘图程序）。我们可以创建这样的交互方式，因为对于所讨论的系统我们只需要使用它的一个子集。

6.3 学习Facade模式

我曾经以外聘开发人员身份为一家大型设计和制造公司工作。我上班的第一天，项目的技术负责人没有到。客户当然不愿意在按小时付工资的情况下，我却无事可做。他们想让我干点什么，就算没多大用处！你是否也有过这样的经历？

一个引导性实例：学会如何使用这个复杂的系统

于是，一位项目成员给我找了些事情。她说："你肯定要学习我们有时会用到的CAD/CAM系统，所以最好现在就开始。从那边的手册开始好了。"然后她带我走到一堆文档前。我可绝没有夸大其词：我要读的手册足有20厘米厚，每页都是21.6厘米×28厘米，而且都用小字印刷！这可真是一个复杂的系统！



图6-1 20厘米厚的手册 = 一个复杂的系统!

我希望与系统隔离开来

现在，如果你、我还有其他四五个人共同开发一个需要使用该系统的项目，应该采用哪一种方式呢？是我们都学习这个系统呢，还是我们抽签决定，输的人负责编写例程，供其他人用来与系统接口？

输的人应该决定，我和团队中的其他人如何使用系统，什么样的应用程序编程接口（API）最适合我们的特殊需要。然后我和程序设计社区的其他人就都能够使用这个新的接口，而无需了解整个复杂的系统了（参见图6-2）。

使用部分功能

这种方法在只使用系统的一部分功能，或者在以特殊方式与系统交互时才有效。如果系统的所有功能都需要使用，那么除非最初的设计很糟，否则并无余地对设计进行改进。

称之谓Facade模式

这就是 Facade 模式（见图 6-3）。通过这个模式我们能够更容易地使用一个复杂的系统，要么只使用系统的一部分功能，要么是以特殊方式使用系统。这里我们的系统就很复杂，但我们只需要使用一部分功能。因此，我们最后得到了一个更简单、更容易使用，或者说按我们的需要量身订做的系统。

不然，

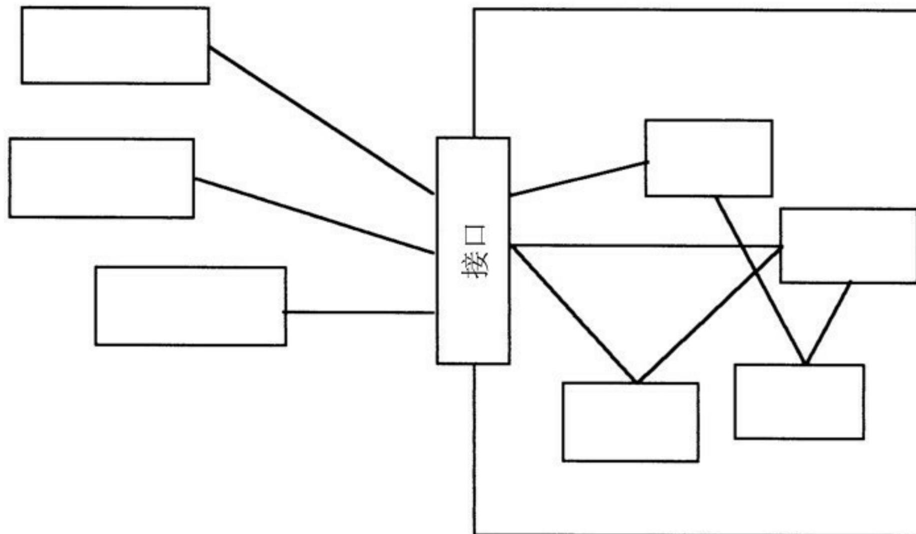


图6-2 将客户与子系统隔离开来

大多数工作还是需要由底层系统完成。Facade模式提供了一组容易理解的方法，这些方法使用底层系统来实现新定义的函数。

Facade模式：关键特征

意图

希望简化原有系统的使用方式。需要定义自己的接口。

问题

只需要使用某个复杂系统的子集，或者，需要以一种特殊的方式与系统交互。

解决方案

Facade为原有系统的客户提供了一个新的接口。

参与者与协作者

为客户提供的一个简化接口，使系统更容易使用。[10]

效果

Facade模式简化了对所需子系统的使用过程。但是，由于Facade并不完整，因此客户可能无法使用某些功能。

实现

定义一个（或多个）具备所需接口的新类。

让新的类使用原有的系统。

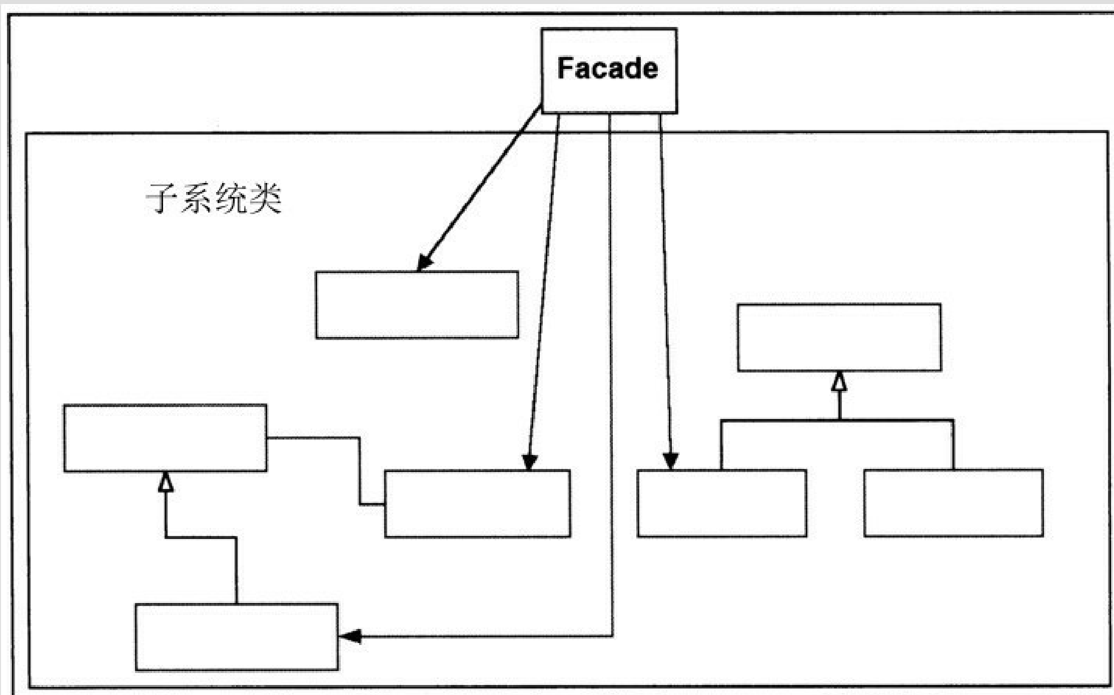


图6-3 Facade模式的通用结构图

6.4 实践注记：Facade模式

Facade模式的变体：减少客户必须处理的对象数量

Facade不仅可以用来通过方法调用创建更简单的接口，还能用来减少客户必须处理的对象数量。例如，假设有一个 Client 对象必须处理Database、Model、Element对象。Client必须首先通过 Database对象打开数据库，获取Model对象，然后再查询Model对象，获取Element对象，最后请求Element对象的信息。如果能够创建一个可供Client查询的Database Facade，那么以上过程将容易得多（参见图6-4）。

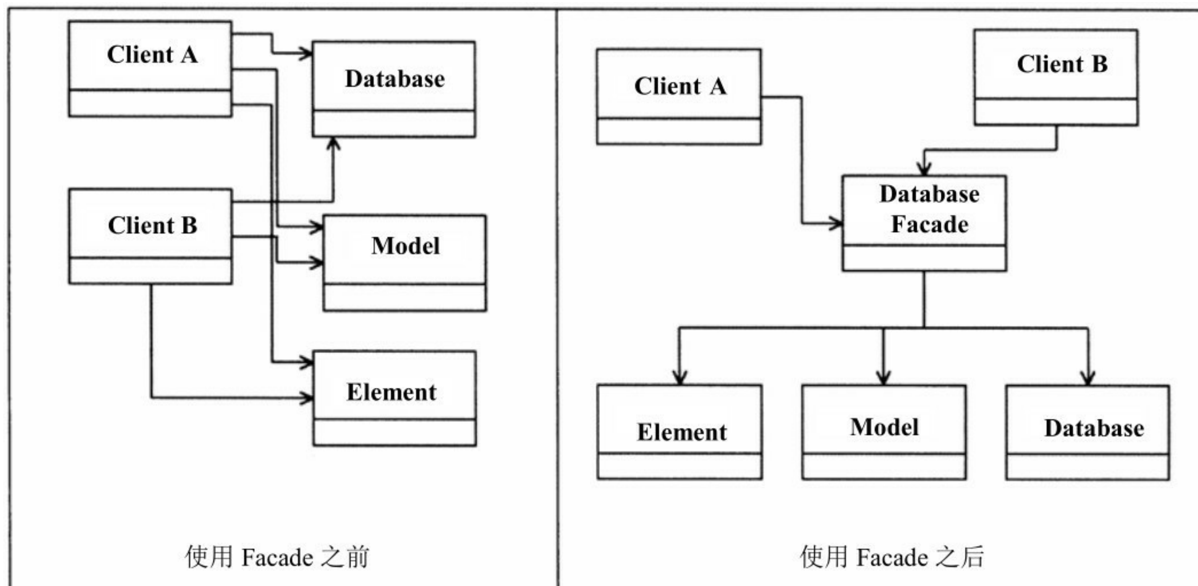


图6-4 Facade模式能够减少客户需要处理的对象数量

让一个Facade为多个对象工作

如果Facade能够设计成无状态的（也就是说，其中没有存储状态），则一个Facade对象就能够被多个其他对象使用。在后面的第21章中，我将讲述如何实现这一点，其中用到了 Singleton 模式和 Double-Checked Locking模式。

Facade模式的变体：用新的例程补充原有功能

假设除了使用系统中原有功能之外，我还需要提供一些新功能——比如，记录对特定例程的所有调用。这种情况下，就不仅仅是使用系统的部分功能了。

这时，我为Facade类所写的方法中可以为新的功能增加一些新例程。这仍然是Facade模式，但是增加了新的功能。我认为其主要目的是简化，因为我不想强制客户例程知道它还需要调用额外的例程——让Facade 去做好了。

Facade 模式提出了一种通用方法；它为我提供了起点。这个模式的Facade部分实际上就是创建了一个新的接口供客户使用，来代替系统的原有接口。我之所以能够这样做，是因为Client对象并不需要原系统提

供的所有功能。

模式提出了通用方法

模式只是提出了通用方法。是否增加新的功能应根据具体情况而定。模式只是可以作为起点的蓝图，可不能完全原样照搬。

Facade模式的变体：一个“封装”层

Facade模式还可以用来隐藏或者封装系统。Facade类能够将系统作为自己的私有成员包含进来。在此情况下，原系统将与Facade类联系起来，但Facade类的客户无需看到。

封装系统的原因包括以下几种。

跟踪系统的使用情况——通过强制所有对系统的访问都必须经过Facade，可以很容易地监视系统的使用情况。

改换系统——未来可能需要切换系统。通过将原系统作为Facade类的一个私有成员，可以最省力地将切换到新的系统。当然，可能还要做很多工作，但是至少我只需在一个地方修改代码（Facade类）就行了。

6.5 Facade模式与CAD/CAM问题的联系

封装V1系统

现在来思考一下第3章中的例子。Facade 模式对于帮助 V1Slot、V1Hole等使用V1System对象是很有用的。我将在第13章中予以实现。

6.6 小结

本章内容

Facade模式之所以如此命名，是因为它在原系统之前放了一个新的

接口（即外观）。

Facade模式可以应用于下述情况。

不需要使用一个复杂系统的所有功能，而且可以创建一个新的类，包含访问系统的所有规则。如果只需要使用系统的部分功能（这是通常的情况），那么你为新类所创建的API将比原系统的API简单得多。

希望封装或者隐藏原系统。

希望使用原系统的功能，而且还希望增加一些新的功能。

编写新类的成本小于所有人学会使用或者未来维护原系统上所需的成本。

复习题

简答题

1.给出Facade的定义。

2.Facade模式的意图是什么？

3.Facade模式的效果是什么？举出一个例子。

4.在 Facade模式中，客户是如何使用子系统的？

5.Facade模式通常能够提供对整个系统的访问吗？

阐述题

1.《设计模式》一书中说：Facade模式是要“为子系统中的一组接口提供一个统一接口。Facade 模式定义了一个更高层的接口，使子系统更加容易使用。”

这是什么意思？

举出一个例子。

2.有一个来自软件之外的Facade实例：有些美国加油站的油泵非常复杂，其上有许多选项，包括如何付款、所用的汽油类型、观看广告等等。给加油泵提供统一接口的方式之一，就是让加油服务员服务。有些

州甚至要求这样。

实际生活中还有类似的能够解释Facade的例子吗？

观点与应用题

1.如果需要在系统所提供的之外添加功能，还能使用Facade模式吗？

2.为什么要使用Facade模式封装整个系统？

3.有什么情况下应该编写一个新系统而不是用Facade封装老系统吗？如果有，请举出。

4.你认为《设计模式》一书中为什么称这个模式为Facade呢？从它的功能来看这个名字合适吗？解释你的回答。

第7章 Adapter模式

7.1 概览

本章内容

我们设计模式学习之旅的第二站，是Adapter（适配器）模式。Adapter模式是一个非常常用的模式，而且你将看到，它可以与其他很多模式结合使用。

在本章中，我们将：

解释Adapter模式是什么，用在何处，以及怎样实现；

给出这个模式的关键特征；

使用这个模式来阐述“多态”的概念；

说明如何在不同的细节层次使用UML；

讲述来自我亲身体验的对 Adapter 模式的一些思考，包括 Adapter模式与Facade模式的比较；

将Adapter模式与CAD/CAM问题联系起来。

7.2 Adapter模式简介

意图：创建新的接口

《设计模式》一书中对Adapter模式的意图是这样叙述的：

将一个类的接口转换成客户希望的另外一个接口。Adapter模式使原本由于接口不兼容而不能一起工作的类可以一起工作。[11]

这段话大致是在说：我们需要一种方式，为一个功能正确但接口不合的对象创建一个新接口。

7.3 学习Adapter模式

一个引导性实例：客户对象无需了解细节

要理解Adapter模式的意图，最简单的方法就是看一个Adapter模式起作用的例子。假设客户给了我如下需求：

为都有“显示”（display）行为的点、线、正方形分别创建类；

客户对象不必知道自己到底拥有点、线还是正方形。它们只需知道拥有这些形状中的一个。

也就是说，我想要用一个更高层次的概念将这些具体形状都涵盖进入，这个高层概念可以称为“可显示的形状”。

在讲述这个简单例子的同时，请想象你曾经遇到的类似情形，比如：

你希望使用其他人编写的子程序或方法，因为你需要它所执行的功能；

你无法将这个子程序直接加入程序中；

子程序的接口或调用方式与需要使用它的相关对象不完全相同。

……这样它就可以以通用的方式处理细节

也就是说，尽管系统中有点、线以及正方形，但我希望客户对象认为只有形状。

这样客户对象可以以相同的方式处理所有对象——无需再关注它们的区别。

这样我未来还可以在客户对象不修改的情况下添加新的形状类型（如图7-1所示）。



图7-1 系统中的对象……应该看起来都是“形状”

怎样实现：多态地使用派生类

我将使用多态，也就是说，我的系统中将有许多不同的对象，但我希望对象的客户与它们的交互方式是通用的。

这里，客户对象只是简单地让点、线或正方形对象进行一些操作，比如“自我显示”或“自我擦除”。然后由每个点、线、正方形负责了解如何按自己的类型完成相应的行为。

为了实现这一点，我创建一个Shape类，然后从它派生出表示点、线、正方形的类（参见图7-2）。

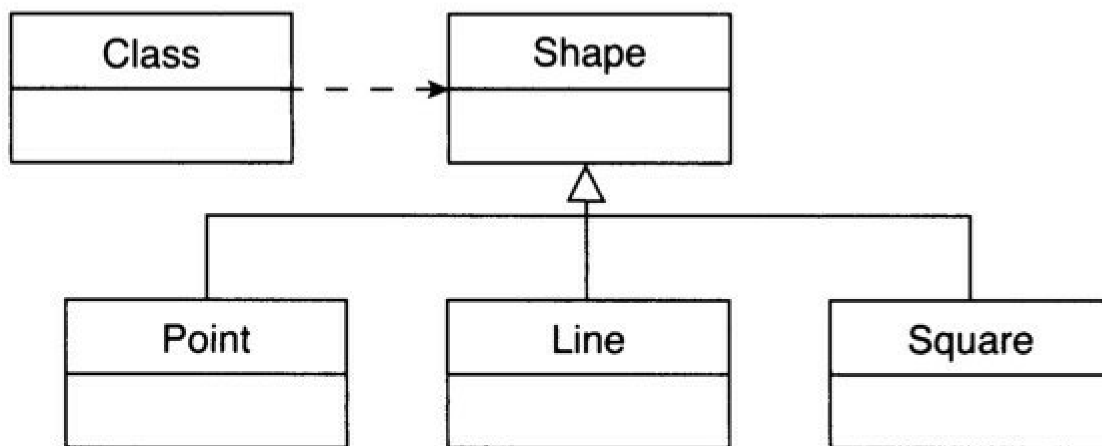


图7-2 Point、Line和Square都是Shape的派生类

本图和本书中所有其他的类图都使用统一建模语言（UML）符号。关于UML符号的叙述，请参见第2章。

怎样实现：定义接口，然后在派生类中实现

首先，我必须指定Shape对象应提供的具体行为。为此，我在Shape类中为这些行为定义了接口，然后在每个派生类中都相应地实现了这些行为。

Shape类需要具备以下行为。

设定一个Shape对象的位置。

获取一个Shape对象的位置。

显示一个Shape对象。

填充一个Shape对象。

设置一个Shape对象的颜色。

擦除一个Shape对象。

如图7-3中所示。

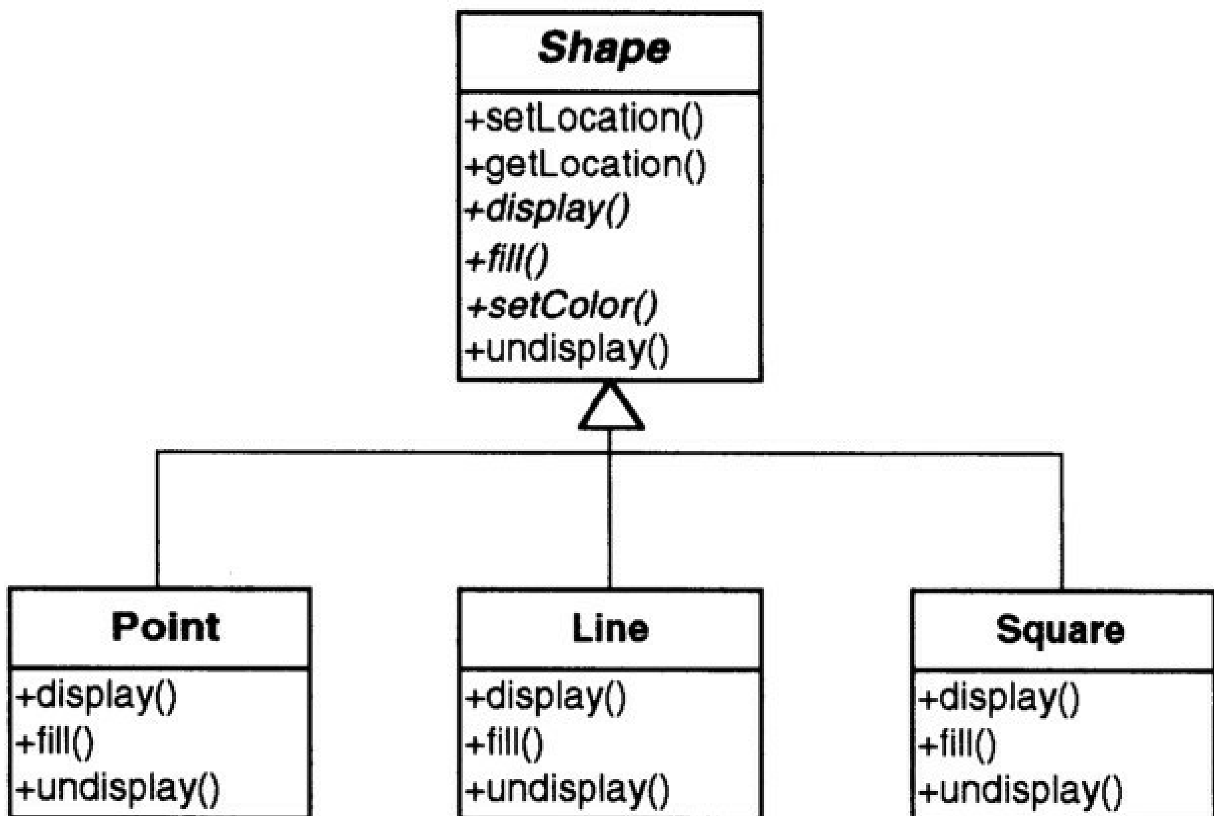


图7-3 显示了方法的Point、Line和Square

现在，增加一种新形状

假设现在客户要求我实现一个圆——一种新的Shape（请记住，需要总在变化!）。为此，我创建一个新的类——Circle类来实现“圆”形，并从Shape类派生出Circle类，这样我仍然可以获得多态行为。

……但是要使用外部的行为

现在，我面临的任务是必须为Circle类编写display、fill和undisplay方法。这可不轻松。

幸运的是，在四处寻找替代方案（优秀的编程人员都应如此）时，我发现大厅那头的 Jill 已经编写了一个处理圆形的类，名叫XXCircle（见图 7-4）。然而糟糕的是，她并没有问过我应该如何命名这些方法。她将这些方法命名为：

displayIt

fillIt

undisplayIt

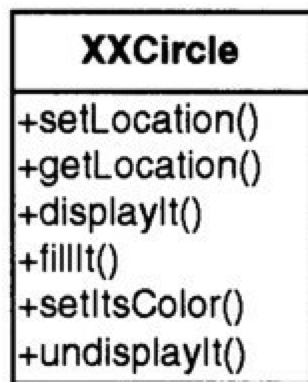


图7-4 Jill的XXCircle类

我不能直接使用XXCircle类

我不能直接使用XXCircle，因为我想保持Shape类的多态行为。这有两个原因：

名称和参数列表不同——XXCircle类中的方法名称和参数列表都和Shape类的不同；

我无法派生它——不但方法名称必须相同，这个类还必须从Shape类派生。

Jill 不可能允许我改变其方法的名称或者从 Shape类派生 XXCircle类。因为为此她需要修改所有正在使用XXCircle的其他对象，而且，我还担心修改别人的代码时，会出现意料之外的副作用。

需要的东西近在眼前，可是却不能使用，而我又不想重写一个。该怎么办呢？

既然不能改变，那就想办法适配吧。

我可以创建一个新类，它就是派生自 Shape 类，因此实现了 Shape 的接口，但是又用不着重写XXCircle类中圆形的实现代码（参见图7-5）：

Circle类派生自Shape；

Circle包含XXCircle；

Circle将发给自己的请求传给XXCircle对象。

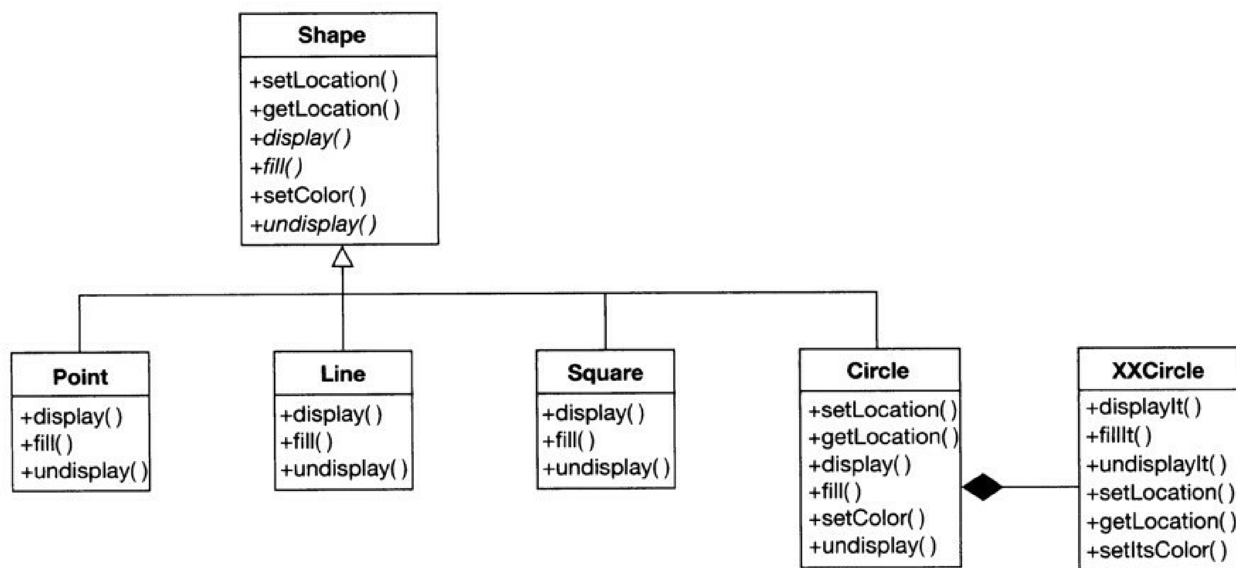


图7-5 Adapter模式：Circle类“包装”了XXCircle类

实现方式

在图7-5中，Circle类和XXCircle类之间直线末端的菱形表示Circle包

含一个XXCircle。当一个Circle对象实例化时，它必须实例化一个对应的 XXCircle对象。发给 Circle对象的任何请求都将转给该XXCircle对象。如果能够总是如此，而且XXCircle对象具有Circle对象所需要的全部功能（等一下我将讨论这种条件不成立时情况如何），Circle对象就可以通过让XXCircle做实际工作来实现自己的行为。

例7-1所示为一个包装的例子。

例**7-1 Java**代码片段：实现**Adapter**模式

```
class Circle extends Shape {
   ...
   private XXCircle myXXCircle;
   ...
   public Circle () {
      myXXCircle= new XXCircle();
   }
   void public display() {
      myXXCircle.displayIt();
   }
   ...
}
```

所获得的效果

通过使用Adapter模式，我可以继续多态地使用Shape类。也就是说，Shape的客户对象不用知道Shape对象实际上代表的是什么类型的形状。这也是能够体现我们对封装新的思考方式的一个例子——Shape 类封装了具体的形状。Adapter 模式最常见的用途就是保持多态性。在后面的章节中将会看到，它常常被用来保持其他设计模式所需要的多态。

Adapter模式：关键特征

意图

使控制范围之外的一个原有对象与某个接口匹配。

问题

系统的数据和行为都正确，但接口不符。通常用于必须从抽象类派生时。

解决方案

Adapter模式提供了具有所需接口的包装类。

参与者与协作者

Adapter改变了 Adaptee的接口，使 Adaptee与 Adapter的基类Target匹配。这样Client就可以使用Adaptee了，好像它是Target类型。

效果

Adapter模式使原有对象能够适应新的类结构，不受其接口的限制。

实现

将原有类包含在另一个类之中。让包含类与需要的接口匹配，调用被包容类的方法。

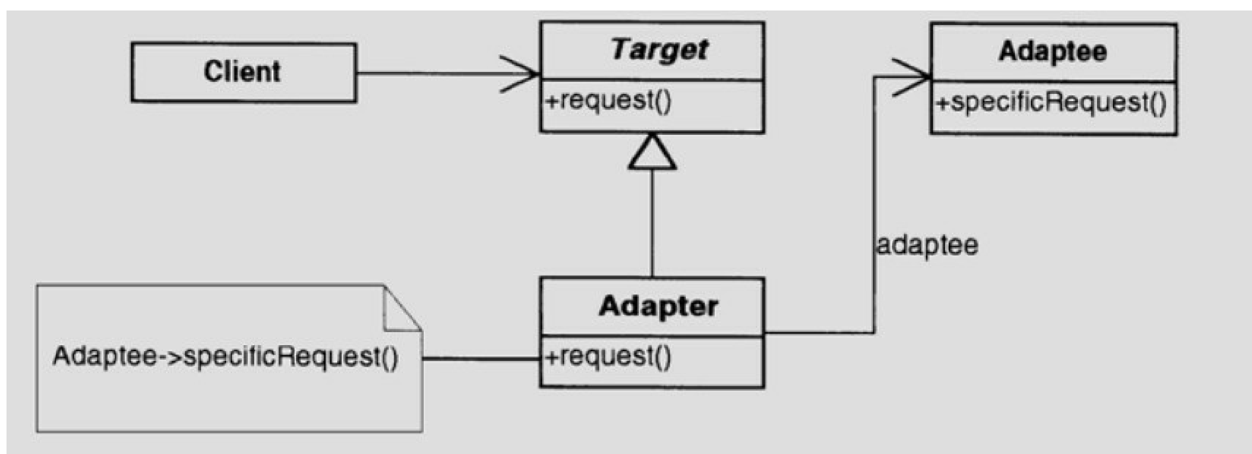


图7-6 Adapter模式的通用结构图[12]

7.4 实践注记：Adapter模式

不只是“包装”而已

我常常会遇到与上述相似的情况，但所要适配的对象可能并不能完成所需的所有功能。

在这种情况下，仍然可以使用Adapter模式，但它并不完全合适。因为这时：

原有类中实现的功能可以适配；

原有类中没有实现的功能可以在包装类中实现。

这当然不能提供与前一种情况同样的好处，但至少用不着实现需要的所有功能。

有了Adapter模式，我用不着担心原有接口了

使用Adapter模式，我在进行设计时就不用操心原有类的接口了。如果有一个类，能够完成所需功能至少在概念上完成了，那么我知道总是可以用Adapter模式为它提供正确的接口。

当你学会更多模式之后，这将变得更加重要。许多模式都要求某些类从同一个类中派生。如果本来就存在一些类，可以用Adapter模式使其适配适当的抽象类（与Circle使XXCircle与Shape相适配一样）。

两种变体：对象Adapter 模式、类Adapter模式

实际上Adapter模式有两种类型。

对象 **Adapter** 模式——前面已经用过的 Adapter 模式称为对象Adapter模式，因为它依赖于一个对象（适配对象）包含另一对象（被适配对象）。

类**Adapter**模式——另一种实现Adapter模式的方式是通过多重继承。这种情况下的Adapter模式称为类Adapter模式。

类Adapter模式的工作原理是创建一个新类，该类同时从两个类继承：

从定义其接口的抽象类公开继承。

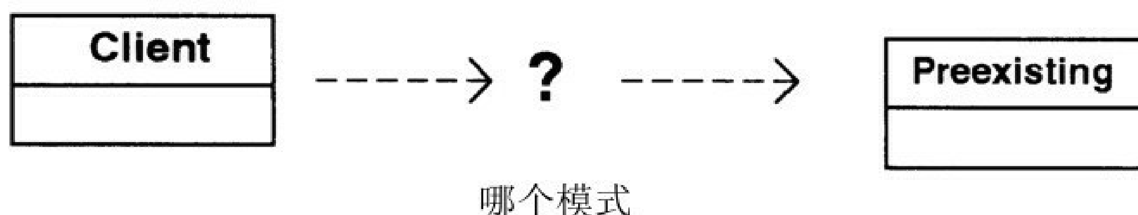
从访问其实现的原有类私有继承。

每个被包装方法都调用其对应的私有继承的方法。

至于应该选择使用哪种Adapter模式，取决于问题领域中实际的不同因素。概念层次上，我可以忽略这些不同；但是，在开始着手实现时，需要考虑所涉及的更多因素。[13]

在我的设计模式课上，几乎总有有人会说："听上去 Adapter 模式和Facade模式好像没什么不同。在两种模式中，都有一个（或多个）已有的类不具备所需的接口。在两种模式中，我都创建了一个具有所需接口的新对象（见图7-7）。"

Adapter模式与Facade模式的比较



哪个模式

图7-7 Client对象使用另一个已有对象，后者接口是错误的

都是包装

包装（wrapper）和对象包装（object　wrapper）这两个术语大家应该都耳熟能详了吧。在考虑用对象将遗留系统包装起来，使其更容易使用时，这种思路是非常常用的。

从这种层次上来看，Facade模式和Adapter模式的确很类似。它们都是包装。但是，它们是不同类型的包装。应该理解它们之间差异，这种差异可能相当微妙。找到并理解这些更加微妙的差异，将获得对模式本质的更深认识。如果要讨论和在文档中描述一个设计，了解这些差异还有助于使其他人也能精确地了解其中各个对象的来龙去脉。让我们看看这两个模式涉及的一些不同因素（见表7-1）。

表7-1 Facade模式与Adapter模式的比较

	Facade 模式	Adapter 模式
是否存在既有的类?	是	是
是否必须按某个接口设计?	否	是
对象需要多态行为吗?	否	可能
需要更简单的接口吗?	是	否

从表7-l中我们能够看到以下内容。

在两个模式中，都存在既有的类。

但是在Facade模式中，我无须按某个接口进行设计；而在Adapter模式中，则必须按某个特定接口进行设计。

在 Facade 模式中我不需要多态行为，而在 Adapter 模式中多态行为可能是需要的。（在某些时候，我只能按特定接口进行设计，那么就必须使用Adapter模式。在这种情况下，多态可能不是问题所在——这就是我说“可能”的原因）。

Facade模式中的动机是简化接口。而在Adapter模式中，尽管也是越简单越好，但是设计必须遵循一个已有的接口，不能简化任何东西，即使可能存在更简单的接口。

并非所有差异都是模式本身的特点

有时候人们可能得出这样的结论：Facade模式与Adapter模式之间的另一个差异，就是Facade隐藏了多个类，而Adapter只隐藏了一个。尽管这种说法经常是成立的，但并不是模式本身的特点。将 Facade 置于一个非常复杂的对象之前，而用Adapter来包装几个共同实现所需功能的小对象，也是可能的。

结论：Facade模式简化了接口，而Adapter模式则将一个已有的接口转换成另一个接口。

7.5 Adapter模式与CAD/CAM问题的联系

在CAD/CAM问题中（见第3章），V2模型中的部件是用

OOGFeature对象表示的。糟糕的是，这些对象的接口不对（在我看来），因为它们不是我设计的。

通过Adapter模式我可以与OOGFeature对象通信

我不能让它们从Feature类派生。但是当我使用V2系统时，它们却可以圆满地完成任务。

在这种情况下，不可能选择编写一个新类来实现这些功能——我必须与OOGFeature对象通信。最简单的实现方案就是使用Adapter模式。

7.6 小结

本章内容

Adapter 模式是一个很常用的模式，它将一个（或多个）类的接口转换成我们需要类所具备的另一个接口。它的实现方式是：创建一个具备所需接口的新类，然后包装原有类的方法，这样实际上就包含了被适配的对象。

复习题

简答题

1.给出Adapter的定义。

2.Adapter模式的意图是什么？

3.Adapter 模式的效果是什么？举出一个例子。

4.定义Shape与Point、Line和Square之间的关系应该使用什么面向对象概念？

5.Adapter模式的最常见的用法是什么？

6.Adapter 模式可以使你不用操心什么方面？

7.Adapter模式的两种变体是什么？

阐述题

1.《设计模式》一书中说：Adapter模式的意图是“将一个类的接口转换成客户希望的另外一个接口。Adapter模式使原本由于接口不兼容而不能一起工作的类可以一起工作。”

这是什么意思？

举出一个例子。

2.“Circle类包装了XXCircle类。”这是什么意思？

3.Facade模式和Adapter模式可能看上去很相似。两者的本质区别在哪里？

4.有一个来自软件之外的Adapter实例：联合国的一个翻译要使来自不同国家的外交官能够用各自的语言阐述和辩论各自国家的立场。翻译采用了一种语言到另一种语言的“动态等价”表示法，这样各种概念将以接受者所期望和需要的方式进行交流。

实际生活中还有类似的能够解释Adapter的例子吗？

观点与应用题

1.什么时候使用 Facade 模式比 Adapter 模式更合适？什么时候使用Adapter模式比Facade模式更合适呢？

2.为什么称这个模式为 Adapter 呢？从它的功能来看这个名字合适吗？解释你的答案。

第8章 开拓视野

8.1 概览

本章内容

前面的章节中讨论了面向对象设计的3个基本概念：对象、封装和抽象类。设计人员对这些概念的看法是非常重要的。糟糕的是，传统的看法有很大的局限性。

本章中，我将回顾并反思在本书前面讨论过的一些主题，同时介绍一些新主题。目的是描述一种看待面向对象设计的全新方式——从理解设计模式的角度出发。然后我将叙述高质量代码的本质品质。这些品质正是敏捷编程方法（如极限编程和测试驱动开发）的提倡者们特别强调的。有趣的是，这些品质在设计模式中也同样存在，而且在遵循设计模式的原则和方法时会“其义自现”。我希望通过从敏捷编程方法和设计模式两种角度阐述这些品质，能够跨越不同设计方法之间的鸿沟。

在本章中，我们将：

比较传统上对对象的看法（将对象视为数据和方法的简单集合）与新的看法（将对象视为具有责任的东西）的异同；

比较传统上对封装的看法（将封装视为数据的隐藏）与新的看法（将封装视为隐藏一切的一种能力）的异同。尤其重要的是，看到封装可以用来包含行为中的变化；

比较对行为变化的不同处理方式的异同；

比较传统的使用继承的方式（用于特化和复用）与新的方式（作为对象分类的一种方法）的异同。新的观点考虑到了包含对象的行为变化；

叙述共性与可变性分析；

说明概念视角、规约视角和实现视角与抽象类及其派生类的关系；

比较设计模式与敏捷编程方法的区别。虽然这些方法一开始看上去互不相容，但是它们实际上是在提倡一些相同的编程特性，包括冗余性、可读性和可测试性。

致谢

也许本章中所述的这种新看法看上去并不是那么有原创性。但我相信，这种看法正是那些最常用的设计模式的发现者在找到最终成为模式的设计时所持的看法。这种看法肯定也是与Christopher Alexander、Jim Coplien（他的著作我将在稍后引用）和GoF的著作一脉相承的。[14]

虽然这种看法可能不那么具有原创性，但是此前还没有人以我在本章和本书中采取的方式对其进行表述过。这种看待设计模式的方式，是我从设计模式本身的行为方式和其他人描述设计模式的方式中提炼出来的。

称之为“新”观点，是指对于大多数开发人员而言它都可能是看待面向对象的新方式。反正在我第一次学习设计模式时，对我来说这确实是全新的。

8.2 对象：传统看法与新看法

传统看法：具有方法的数据

传统上对象被视为具有方法的数据。按这种观点，我的一个老师曾将对象直呼为“智能数据”。在他看来，这只是一种处理数据的智能方式而已：“开始时是描述问题领域中状态的数据，然后添加处理数据的方法（必要的行为所需要的），瞧，对象就有了！”可是这也太简单、太肤浅了。这其实只是从实现的视角来看待对象而已。

新看法：具有责任的实体

更有意义的定义应该是从概念视角出发——对象是具有责任的一个实体。这些责任定义了对象的行为。有时我还将对象视为具有特定行为的实体。

这个定义显然更好，因为它有助于使我们关注对象的意图行为，而不是对象如何实现。这种理解使我能够以两个步骤构建软件。

1.先做出一个初步的设计，不用操心所有的相关细节。

2.实现该设计。

关注对象要做什么，还能帮助我免于过早地操心实现细节，从而将这些实现细节隐藏起来。这进而能够帮助我构建出未来更加容易修改的软件……如果我需要修改的话。

这种方式之所以能够行之有效，是因为我只需关注对象的公开接口——这是我要求对象完成某些工作的交流渠道。有了好的接口，我能够要求对象完成其责任范围之内的任何工作，而且可以相信它能够完成，我不需要知道在对象内到底是怎样运作的，不需要知道它怎样处理我传递给它的信息，也不需要知道它怎样收集其他需要的信息。我大可以放心地全权委托给它。

例如，假设我有一个Shape对象，它的责任如下：

知道自己的位置；

能够在显示器上绘制自己；

能够从显示器上擦除自己。

这些责任意味着必须存在如下方法：

getLocation(...)

drawShape(...)

unDrawShape(...)

但是对于Shape对象内的任何情况，我们未置一词。我只关心Shape将对自己的行为负责。它的内部可能有一些属性，可能有一些执行计算的方法，甚至可能引用其他对象。Shape对象可能包含有关自身位置的

属性，也可能引用另一个对象（例如，从数据库中）获得自身位置。这为你提供了满足建模目标（或者在目标改变时修改代码）所需的灵活性。

关注动机而非实现，是设计模式中反复出现的主题，当然，这很容易理解。因为将实现隐藏在接口之后，实际上是将对象的实现与使用它们的对象解耦了。

以这样的方式看待对象吧。让它成为你对对象的基本观点。你将因此而得到优秀的设计。

8.3 封装：传统看法与新看法

我的面向对象伞

在我的面向模式设计课程中，我经常问学生们："有谁听说过封装的定义是'数据隐藏'？"几乎每个人都会举手。

这时我会开始讲有关我的伞的故事。我回忆起自己在西雅图的经历，那是一个多雨的城市，虽然没有宣传的那么厉害，但在秋、冬和春季，那里的雨确实很多。在西雅图，雨伞和带帽上衣每个人都是离不开的!

让我来说说我的大伞的故事吧。它真的非常大！实际上，除我之外，还可以再容纳三四个人。当我们在这把伞下躲雨时，还可以把它从一个地方移动到另一个地方。它还有立体声音响系统，除了为我们遮雨之外还可以提供娱乐。更令人惊讶的是，它还可以调节空气的温度，使我们更温暖或更凉快。这是一把多么酷的伞啊！

我的伞很方便。它就在那里等着我。它还有轮子，因此我用不着拎着它到处走。我甚至用不着推着它，因为它可以自己驱动。有时候我还可以打开伞顶，晒晒太阳。（为什么在阳光灿烂的时候我还要用伞呢？这我可无可奉告。）

在西雅图，有几十万把这样的伞，它们五颜六色。

大多数人称它们为汽车。

但我确实将我的汽车看成伞，因为伞就是我用来避雨的东西。很多次，当我在室外等待与什么人见面时，我就坐在“伞”里，免得淋湿！

定义可能有局限

当然，汽车并不是真正的伞。是的，你可以用它来避雨，但这样看待汽车的局限性太大了。同样，封装也不仅仅是数据隐藏，这样看待封装的局限性太大了。这样的思考方式，会限制我在设计时的思路。

封装应该被视为“任何形式的隐藏”。换句话说，可以是隐藏数据，但还可以是隐藏以下各种东西：

实现细节；

派生类；

设计细节；

实例化规则。

如何看待封装

前面讨论隐藏实现时，我实际上是在“封装”实现细节。现在更进一步，考虑图 8-1 所示的类图，它出自第7章。其中有均属 Shape类型的Point、Line、Square和 Circle。而且 Circle“包装”或者说包含了XXCircle。

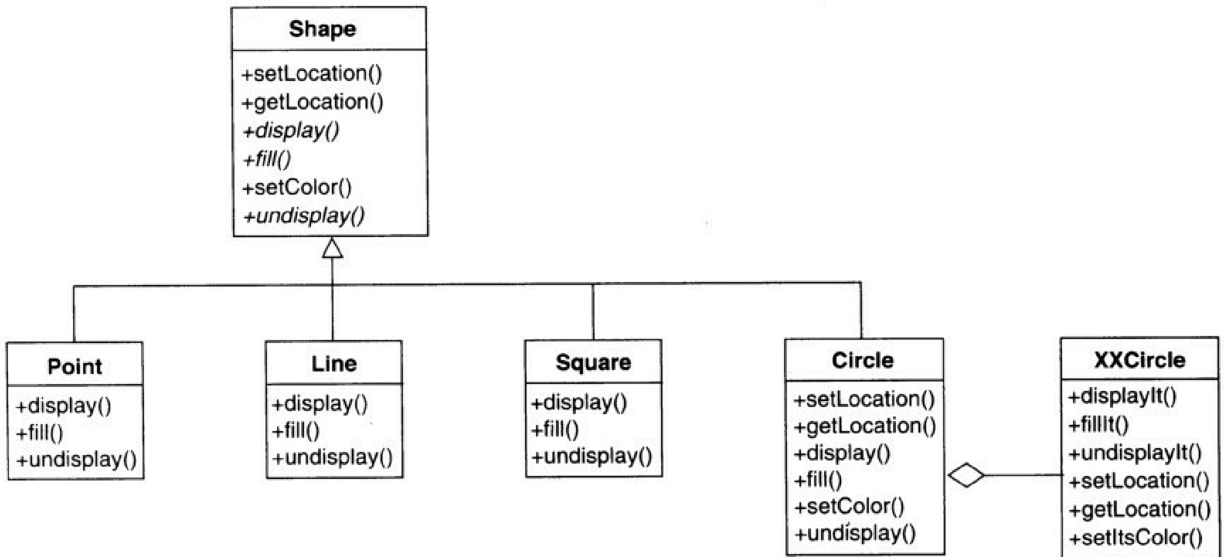


图8-1 用Circle适配XXCirc1e

图8-1中有很多种类的封装。

封装的多种层次

数据的封装——Point、Line、Square及 Circle对象中的数据对其他所有对象都是隐藏的。

方法的封装——如Circle类的setLocation方法。

其他对象的封装——除Circle之外，其他对象对XXCircle对象都一无所知。

类型的封装——Shape类的客户代码看不到Point、Line、Square或Circle。

类型的封装是通过多态使用具有派生类的抽象类（或者具有多种实现的接口）实现的。使用该抽象类的客户代码无需知道派生类的实际类型，这正是《设计模式》一书中封装的通常含义。

这种新定义的优点

以这种更宽泛的方式看待封装，其优点是能够带来一种更好地切分（即分解）程序的方法。封装层就成为设计需要遵循的接口。通过封装Shape的不同派生类，我可以不修改使用它们的任何客户程序，而增加

新的派生类。通过将 XXCircle封装于 Circle之后，我可以在未来改变这种实现方式——如果选择或需要这样做。

作为概念的继承与用来复用的继承

面向对象范型的早期提倡者曾将“类的复用”作为巨大优势之一大力鼓吹。这种复用通常是通过先创建基类，然后从这些基类出发派生新类而实现的。因此，为这些从其他类（称为泛化类）派生的子类专门定了一个术语：特化类。

我不想讨论这个术语，相反，我是要提出另外一种使用继承的方式，我认为它的功能更加强大。例如，假设我需要使用五角形。我定义了一个Pentagon 类，它包含状态、绘制操作、擦除操作等等。后来我又发现需要一种带有特殊边线的五角形，这时我可以从最开始的Pentagon类派生一个新的、“特化的”Pentagon类，它绘制边线的方式不同（见图8-2）。

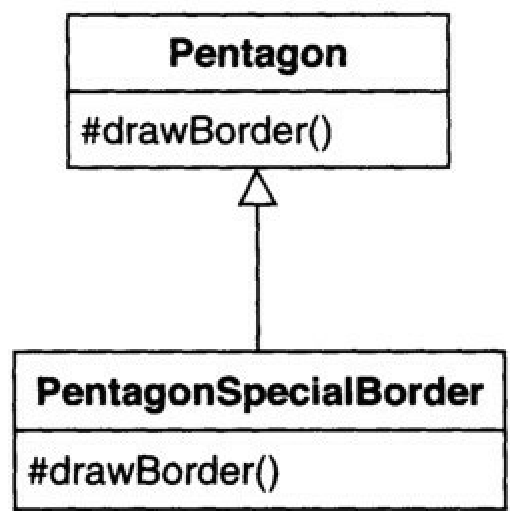


图8-2 从Pentagon派生的PentagonSpecialBorder

这是一个使用继承来特化的例子。我复用了 Pentagon，得到PentagonSpecialBorder。这种方式非常行之有效，但是存在三个问题，如表8-1中所述。

表8-1 使用继承来特化的问题

此方式的问题	说　　明
可能导致弱内聚	思考一下如果有许多许多种边线类型，情况会怎么样。那样的话，Pentagon 不能只关心五角形了，它们还要关心所有不同的边线绘制——这将使类不得不处理更多问题。我还想到了 Pentagon（及其派生类）的其他东西会变化（比如其中用到的模式）
减少复用的可能性	当我为不同边线编写了代码，并加入 Pentagon 及其派生类中，怎样才能将这些代码复用于其他形状呢？这可能是非常困难的，因为每次复用上下文都在变化，而且完成这项功能的代码是在 Pentagon 中，其他地方不太可能轻易地访问它们
无法根据变化很好地伸缩	这种为了复用的特化技术在课堂中很有效，因为在展示它并继续讲其他内容的时候，不会有人询问这样的问题："如果其他情况开始发生变化，怎么办？"例如，如果有两种不同的底纹怎么办？为了应对所有选择，需要不断地特化（因此会部分重复）Pentagon 类

另一种使用继承的方式，是将类按相同行为分类。我将很快进一步讨论这一点。

8.4 发现变化并将其封装

在《设计模式：可复用面向对象软件的基础》一书中是如下建议的：

设计模式用继承对行为变化进行分类

考虑你的设计中哪些地方可能变化。这种方式与关注会导致重新设计的原因相反。它不是考虑什么会迫使你的设计改变，而是考虑你怎样才能够在不重新设计的情况下进行改变。这里的关键在于封装发生变化的概念，这是许多设计模式的主题。[15]

对此我喜欢的说法是："发现变化并将其封装。"

如果你仅仅将封装视为数据隐藏的话，这些说法可能看起来有些奇怪。而当你将封装看成是通过抽象类或者接口隐藏类——"类型封装"[16]时，它们就合理多了。包含这种抽象类或者接口类型的引用（也就是聚集），可以隐藏表示行为变化的派生类。也就是说，使用它们的类引用抽象类或者接口，而后者具有多个特化的派生类。这些派生类对使用它们的类而言是隐藏的（也就是封装了的）。

实际上，很多设计模式都使用封装在对象之间创建层。这样设计者可以在层的两侧进行修改，而不会对另一侧产生不良影响。这有利于两侧的松耦合。

包含数据变化与包含行为变化

这种思考方式在Bridge模式（将在第10章“Bridge模式”中讨论）中非常重要。但是，接下来我们暂时不继续讨论这一主题，而是转而说明许多开发者都持有的一种设计偏见。

假设我正在从事这样的一个项目：对动物的不同特征建模。项目的需求如下：

每种动物都有数量不同的腿；

动物对象必须能够记住并获取这一信息；

每种动物的移动方式都不同；

对于给定的地形类型，动物对象必须能够返回从一个地方移动到另一个地方所花费的时间。

要处理“腿数”的变化，典型的方式是用一个数据成员存放这个值，还有一些方法来设置和获取这个数据成员的值。但是，处理行为上的变化，通常采取另一种方式。

假设有两种不同的移动方式：行走和飞翔。满足这样的需求，需要通过两种不同的代码：一种是既处理行走，又处理飞翔；一种是只用一个简单的变量，但是无法用于整个解决方案（虽然可以用来表示有哪种移动方式）。既然存在两种不同的方法，我将不得不面临设计上的抉择：

用一个数据成员说明我的对象拥有哪种移动方式；

用两种不同类型的Animal类（都派生自Animal基类）——一个表示能够行走，另一个表示能够飞翔。

糟糕的是，当问题更加复杂时这两种方式都不可行。虽然两种方法在只有一种变化时（移动方式），都能够奏效，但是如果存在更多变化时会怎么样呢？例如，如果我们还有老鹰（会飞翔的肉食动物）、狮子（会行走的肉食动物）、麻雀（会飞翔的食草动物）和牛（会行走的食草动物），该怎么办？使用一个开关变量表示动物的类型，将移动方式

和食性结合起来——这两者实际上没有太多关系。或者对每一种特殊情况都使用继承，生成许多类。同样，如果动物在某些情况下会表现出一种行为，而在另一些情况下会表现一种行为，又该怎么办呢（大多数鸟类既会行走，也会飞翔）？

另一个问题又出来了。当一个类处理越来越多的不同变化（比如通过开关变量）时，代码的内聚性会变得很差。也就是说，它所处理的特殊情况越多，可理解性就越差。

用对象处理行为上的变化

此外还存在着另一种可能性：让Animal类包含一个具有合适移动行为的对象，如图8-3所示。

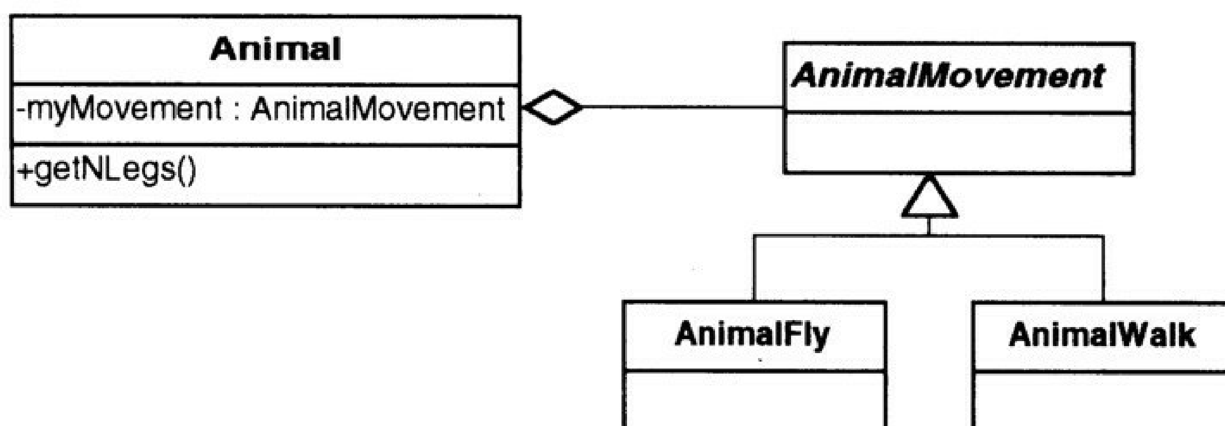


图8-3 Animal对象包含AnimalMovement对象

这可不是小题大做

这可能看上去有些小题大做。但是，这其实也不过就是Animal对象中包含了一个数据成员而已，只不过这个数据成员是包含了Animal移动方式的对象。这与用一个成员存放腿的数量（其中是用一个内置类型的对象存放腿的数量）非常类似。我怀疑这里概念上的差异可能远甚于实际上的差异，因为图8-3和图8-4看上去很不一样。

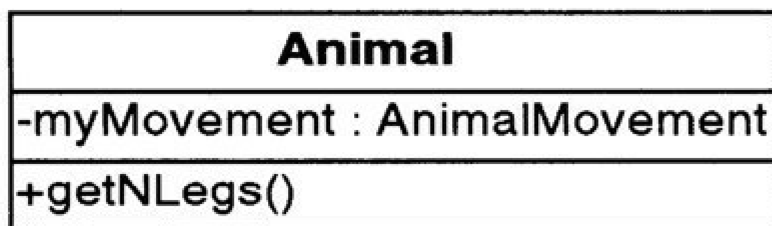


图8-4 用一个成员表示“包含”关系

两者的比较

许多开发人员往往会认为“一个包含了另一个对象的对象”本质上与“一个具有纯数据成员的对象”完全不同。但是那些看似不是对象的数据成员（例如，整型数和双精度数）实际上都是对象。在真正的面向对象语言中，万事万物皆对象，甚至内置数据类型也是对象，即使其行为只是算术运算。这些对象的特殊语法（例如，x+y就等同于x.addTo(y)）掩盖了这样的事实：它们在行为上其实也是对象。

用对象的属性来包含变化，和用对象的行为包含变化其实非常相似。通过一个例子可以非常容易地说明这一点。假设我要编写一个电子收款机系统。在这个系统中，需要有销售收据。在收据上有总计一项。我一开始用double类型表示总计。但是，我所要处理的应用程序将要用在世界各地，我很快意识到需要解决货币兑换等问题。因此需要创建一个 Money类，用来包含数量和货币种类。现在总计变成一个Money类型的了。

这样使用 Money 类，看起来似乎只是在使用一个包含更多数据的对象。然而，当我需要将某个Money对象从一种货币兑换成另一种货币时，进行兑换的将是Money对象自己，因为对象应该对自己负责。一开始，好像转换只需要再设另一个数据成员来指定兑换汇率是多少，就可以完成了。

可是，情况可能会更加复杂。例如，可能我需要能够根据日期来兑换货币。我可以将货币属性改成一个 Currency 类。如果在 Money 类或

Currency类中添加行为，SalesReceipt（销售收据）类中其实也就根据所包含的Money对象（从而包含的Currency对象）的具体情况添加了相应的行为。但是，这样做并不会使SalesReceipt类更复杂，也不会要求对SalesReceipt类进行任何修改。

我将在后面的几个设计模式中，说明这种通过使用包含对象来执行所需行为的策略。

8.5 共性和可变性分析与抽象类

共性分析和可变性分析

Jim Coplien有关共性和可变性分析的工作，告诉了我们如何在问题领域中找到不同变化，如何找到不同领域中的共同点：找到变化的地点，称为“共性分析”；然后找出如何变化，称为“变性分析”。

共性分析

按照Coplien的说法：“共性分析就是寻找一些共同的要素，它们能够帮助我们理解系列成员的共同之处在哪里。”[17]这里的“系列成员”，Coplien用来指通过表现方式或者所执行的功能相互关联的一些要素。找出事物的共同点这一过程将定义这些要素所属的系列（因而也就定义了不同点）。例如，如果我向你展示一支白板记号笔、一支铅笔和一支圆珠笔，你会说它们都具有的共同点在于它们都是“书写工具”。这里你所做的识别其共同点的过程就是共性分析。有了这种共性（书写工具），你就能更容易地讨论，作为书写工具它们有何差异（书写的材料不同，形状不同，等等）。

可变性分析

可变性分析揭示了系列成员之间的不同。可变性只有在给定了共性之后才有意义：

共性分析寻找的是不可能随时间而改变的结构，而可变性分析则要

找到可能变化的结构。可变性分析只在相关联的共性分析定义的上下文中才有意义……从架构的视角来看，共性分析为架构提供长效的要素，而可变性分析则促进它适应实际使用所需。[18]

也就是说，如果变化是问题领域中各个特定的具体情况，共性就定义了问题领域中将这些情况联系起来的概念。共通的概念将用抽象类表示。可变性分析所发现的变化将通过具体类（也就是从抽象类派生而来的具有特定实现的类）实现。

寻找对象的新范型

面向对象设计新手经常被人教导，应该从问题领域中查找入手："找到其中的名词，创建对象表示它们。然后找到与这些名词相关的动词（也就是它们的操作），并通过在对象中添加方法来实现这些动词。"这种集中考虑名词和动词的过程通常会得出比我们预期更大的类层次结构。我建议不要使用这种只看名词和动词的方法，在创建对象时使用共性和可变性分析更好。（这与Coplien的方法是一致的，但并不没有包括其方法的所有内涵。）

思考一下图8-5。它显示了以下几个方面之间的关系。

面向对象设计涵盖所有三种视角

共性和可变性分析。

概念视角、规约视角和实现视角。 .

抽象类及其接口和抽象类的派生类。

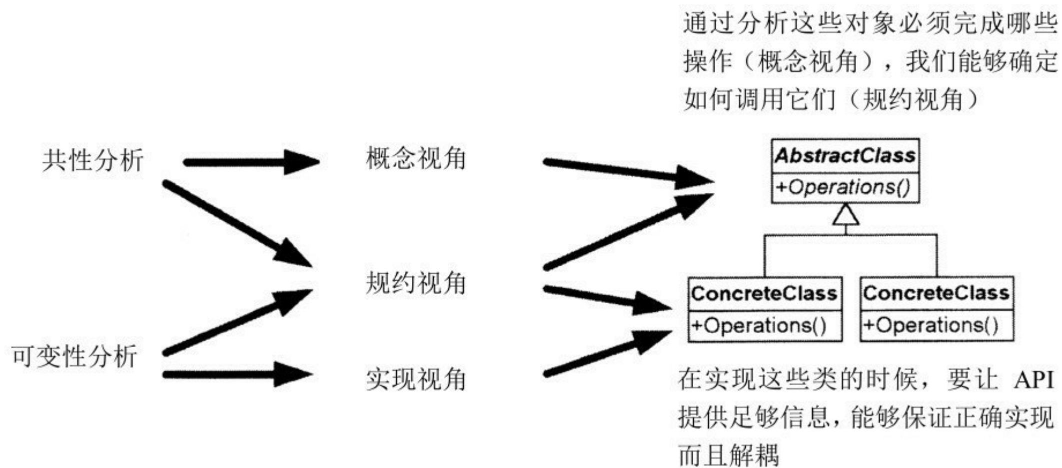


图8-5 共性和可变性分析、三种视角与抽象类之间的关系

从图8-5中可以看到，共性分析与问题领域的概念视角是互相关联的，而可变性分析与（特定情况的）实现是互相关联的。

现在，规约使我们能够更好地理解抽象类

规约视角则位于其中，共性和可变性要涉及这个视角。规约描述了如何与一组概念上相似的对象通信。这些对象每一个都代表了公共概念的一种变化。在实现层次上这个规约将成为抽象类或者接口。

按照我们看待面向对象设计的新方式，现在可以非常自信地说出如下结论（列于表8-2中）。

表8-2 使用抽象类进行特化的好处

与抽象类的对应关系	描　述
抽象类与核心概念	抽象类代表了将所有派生类联系起来的核心概念。正是这个核心概念定义了派生类的共性
共性与要使用的抽象类	共性定义了需要使用的抽象类
可变性与抽象类的派生类	在共性中辨别出的可变性将成为抽象类的派生类
规约与抽象类的接口	这些类的接口对应于规约层次

这样类的设计过程就简化成了两个步骤，如表8-3所示。

表8-3 设计的两步法

定义……时	必须问自己……
抽象类（共性）	需要用什么接口来处理这个类的所有责任
派生类（可变性）	对于这个给定的特定实现（这个变化），应该怎样根据给定的规约来实现它

规约视角和概念视角之间的关系在于：规约标识了用来处理此概念所有情况（即概念视角所定义的共性）所需的接口。

规约视角和实现视角之间的关系在于：对于给定的规约，怎样实现这个特定情况（这个变化）？

8.6 敏捷编程的品质

“预先设计”与“边编程边设计”

设计模式方法经常被人叙述成需要“预先设计”。它提倡从问题领域的一些主要概念着手，然后深入，逐步考虑更多的细节。

在极限编程（eXtreme programming，XP）中提倡另一种方法，看上去好像与设计模式是矛盾的。极限编程的核心是循序渐进地开发，在编程的同时进行验证。大的概念是从众多小的概念中演变出来的。

我认为极限编程和设计模式并不是对立的，相反，它们是相辅相成的关系。它们都能够用来达到同样的目的：高效、健壮和灵活的代码。这怎么可能呢？我相信这是因为两种方法所遵循的原则紧密相关。

敏捷编程中吸取的教益

作为非常早就接受了敏捷编程实践的人，我发现自己处于进退两难的境地：

我使用预先设计已经非常成功；

而敏捷方法却告诉我尽量不要如此（有时候实际上是不能如此）。

我更加成功了。

我的困境在于，我很清楚设计模式对于我的成功至关重要，我不愿意放弃它。但是，我希望遵循的敏捷方法却似乎不建议预先设计。直觉上，我感到它们应该有共通之处：敏捷方法要求代码具有可变性，而模式能够产生灵活的代码。可能这只是方法上的不同，而非道之不同。

最终我解决了这一难题。我认识到，两种方法都要求代码中具备同

样的品质。它们只是殊途同归罢了。不同的代码品质实际上是密切相关的。例如，当方法封装时，它实际上也解耦了。敏捷实践关注的是另一些我还没有提到的品质，但是，这些品质与我提到的密不可分。它们包括：(1) 无冗余；(2) 可读；(3) 可测试（顺序与重要性无关）。

无冗余

要遵循的一个非常重要的策略就是，某个规则只在一个地方实现。长期以来，“一个规则，一个地方”已经成为面向对象设计人员挂在嘴边的口头禅。它代表了设计人员的一个最佳实践。最近，Kent Beck称这一实践为“一次且仅一次规则”。[19]

他将此实践定义为他所谓的“约束”的一部分。

1.系统（代码和测试的合称）必须能够表达要表达的一切。

2.系统必须不含重复代码。（这两点共同构成了“一次且仅一次规则”。）

也就是说，如果对如何行事有什么规则的话，也应该只实施一次。这通常要求使用几个小的方法，所增加的开销是非常小的，而好处却很明显：消除了重复，经常还避免了未来原本可能出现的许多问题。重复是一件很糟糕的事情，不仅因为多次重复输入所带来的额外工作，而且因为在未来有什么需要修改的时候，很可能会漏改一些地方。

我不是纯粹主义者，但是如果说有什么场合我认为总是遵循规则非常重要的话，这就是。冗余与耦合之间往往纠缠不清。如果存在冗余代码，那么修改其中一部分将非常必要，而且很可能另一部分也需要修改。因此，这两部分代码实际上是互相耦合的。

有趣的是，遵循“按接口设计”的做法，找出变化之处，从而使代码高度内聚，正是消除冗余代码所需要的。这是因为，这样能防止同样的代码出现在两个地方。为了避免耦合，代码需要封装在定义明确的接口之后。

可读性

可读性是敏捷方法中提倡的优秀代码的另一种必需的品质，它与强内聚息息相关。但是Ron　Jeffries（极限编程倡导者）提出了“按意图编程”的要求，将这一品质提升到了新的层次。[20]简单而言，这一要求是说，当你编写代码时，需要实现一些函数，只是假定它们已经存在，为它们指定“反映意图的名字”，[21]编写对这些函数的方法调用，然后继续（以后再实现函数）。也就是说，编程变成了一系列对函数的调用，这些函数的命名清晰地说明了它们的用途。

这将产生可读性非常好的代码，因为在更大的模块层次，代码阅读者要看的是代码的意图，而不是所有细节实现。Martin　Fowler对此更进一步，他这样写到：“每次当我们感觉需要为什么东西加注释的时候，相反我们会编写一个方法。”[22]其结果就是更简短、更清晰（而且紧凑）的方法。

按意图编程与设计模式“按接口设计”的要求非常类似。当你在给出“反映意图的名字”时，你就是在不考虑实现的情况下创建接口。同样，这里敏捷编码技术和设计模式在如何达到代码高质量上又是英雄所见略同。

可测试性

可测试的代码是优秀代码的主要品质。而可测试性正是敏捷方法的核心。在讨论这一点之前，我想先澄清“可测试性”与极限编程中“编写代码之前先写单元测试”这一实践的区别。[23]

极限编程中非常独具特色的实践之一，就是在编写代码之前就编写测试。这有几个目的：

最后能得到一组自动化测试；

必须按方法的接口而非实现来设计，这样能够得到封装性更好、耦合更松散的方法；

关注测试会使你注意将概念分成多个可测试的部分，这样能够也能获得强内聚和松耦合。

我称容易测试的代码为可测试代码。可测试代码就是这样的代码：它们能够单独测试，而且无需操心与其他模块或者实体如何耦合。极限编程“预先编写测试”的实践本质上会产生可测试性很高的代码。

可测试性与其他实践是密切相关的。

内聚的代码更容易测试，因为代码只负责一项责任。

松耦合的代码比紧耦合的代码更容易测试，因为需要操心的交互少。

冗余代码本身并不会增加测试的难度，但是需要更多测试覆盖冗余。所以，如果冗余度增加，整个系统的可测试性会降低。

可读性好的代码更容易测试，因为这种代码的方法名和参数都能更精确地说明各自的意图。

封装性好的代码更容易测试，因为它与其他代码没有耦合或者耦合很少。

为了说明这一点，我还是举一个例子吧。有一个客户曾经在我上“高效面向对象分析与设计”课程之前，和我讨论测试问题。他说：“不要老说什么单元测试，我们用过，效果可不怎么样。”我问他怎么回事。他说，在以前的一个项目中他们曾经尝试对代码进行单元测试，但是非常困难。为了编写测试，他们不得不构建了一个框架，因为无法事先实例化要测试的对象。这些对象与其他对象互相纠缠在一起了。

我反问他，他们是否考虑过应该在编写代码之前就测试函数。我的客户说，他们没有想过。我又问他，如果先考虑以后如何测试代码，他们代码的设计是否会有不同。他停顿了一下，认识到如果事先考虑以后对代码的测试，确实能够改善他们的设计。

许多开发人员将这种理念继续推进，通过测试驱动整个开发过程，这种方法就称为测试驱动开发（Test-Driven Development，TDD）。这已经超出了本书的范围。[24]我对测试驱动开发有过不少经验，深信这

是一种非常好的方法。与其他敏捷方法一样，测试驱动开发乍看起来似乎与模式方法互不相容，实则不然。它与模式基于相同的原则，只是处理代码编写任务的方式不同罢了。

8.7 小结

本章内容

传统上看待对象、封装和继承的方式有很大局限性。封装的存在绝不仅仅是为了隐藏数据。将其定义扩展为“任何形式的隐藏”之后，我可以用封装来创建对象之间的层——这样我就可以对位于层一侧的东西进行修改，而不会对另一侧造成不良影响。

将继承视作一种一致地处理概念上相同的各个具体类的方法，比看成一种特化方法，要合理得多。

使用对象保存行为变化的概念，与使用数据成员保存数据变化的做法不同，但是两者都考虑到了所保存的数据或者行为的封装（以及扩展）。

用共性和可变性分析在我们的问题领域寻找对象，比寻找名词和相应的动作更加有效。

敏捷方法，尤其是极限编程所提倡的诸多实践能够产生许多好的代码品质，这些品质与长期以来大家公认的代码品质——强内聚、松耦合和封装紧密相关。

复习题

简答题

1.看待封装的正确方式是什么？

2.看待一个问题的三种视角是什么？（可能需要复习第1章。）

阐述题

1.对对象有两种理解方式："具有方法的数据"和"具有责任的实体"。

第二种方式在哪些方面优于前者？

它揭示了哪些本质？

2.对象能够包含另一个对象吗？这与对象包含一个数据成员有何不同？

3."发现变化并将其封装"这句话是什么意思？举出一个例子。

4.解释共性/可变性分析和看待问题的三种视角之间的关系。

5.抽象类对应于"核心概念"。这是什么意思？

6."可变性分析揭示了系列成员之间的不同。可变性只有在给定了共性之后才有意义。"

这是什么意思？

应该用什么样的对象表示共通的概念？

应该用什么样的对象表示变化？

观点与应用题

1.为什么开始把注意力集中放在目的上比放在实现上更好？举一个你自己遇到过的例子说明这一点。

2.先入之见会限制人们对概念的理解，这一点我们在封装中已经看到了。你能够想出自己遇到过什么情况下，先入之见会妨碍对需求的理解吗？请叙述当时的情形，你又是怎么解决的？

3.术语继承可以用于两种场合：从非抽象类派生一个类，以得到特化版本；将一个派生类作为不同实现的起点。如果我们改用两个不同的术语表示这两种概念，是否更好？

4.如何使用共性/可变性分析帮助我们思考修改系统的各种方式？

5.尽早而且经常地探究变化是非常重要的。你认可这种说法吗？说明你的理由。它有助于避免一些问题吗？怎样避免的？

6.共性/可变性分析是寻找对象的一种重要的工具，比“寻找名词”好。你同意这种说法吗？说明你的理由。

7.本章试图阐述一种看待对象的新视角。你认为这一目的达到了吗？说明你的理由。

第9章 Strategy模式

9.1 概览

本章内容

本章将介绍一个新的来自电子商务（通过Internet进行的电子商务）领域的案例研究，同时还将使用Strategy（策略）模式给出一个解决方案。我们到第16章“分析矩阵”时还会谈到这个案例。

在本章中，我们将：

再次讨论新需求的问题，叙述处理新需求变更的途径；

解释哪一种方式与《设计模式》一书中理念一致及其原因；

介绍这个新的案例研究；

描述Strategy模式，并展示这个模式如何处理我们的案例中的新需求；

描述Strategy模式的关键特征。

9.2 处理新需求的一种途径

灾难往往是由短期未臻最优的决策，长期累积而引起的

在生活中，我们经常需要就执行某项任务或者解决某个问题的一般方式做出抉择，这种情况在软件开发中也屡见不鲜。我们大多数人都已经了解到，短期的抄近路，可能会在长期导致问题严重复杂化。例如，我们都不会忘记行驶超过一定里程之后给汽车换机油。是的，我用不着每3 000英里就换，但是决不能到30 000英里再换。（如果真的那样，也就用不着换机油了：汽车已经报销!）再来想想所谓的“桌面文件系

统”——用桌面存放文件的“技术”。短期内当然很不错，可是时间一长，文件越堆越高，要找什么就麻烦了。灾难往往是由短期未臻最优的决策，长期累积而引起的。

在软件开发中也是如此：只关心眼前的事情，而忽视长期问题

糟糕的是，在软件开发领域中，很多人还没有认识到这一点。许多项目只关心处理眼前的紧迫需求，却不顾将来的维护。项目之所以会忽视易维护性或者可修改性等长期问题，其原因无非有如下几个。

我们确实无法预测新需求将如何变化。

如果我们试图预测未来的变化，那么在分析阶段就会止步不前。

如果我们要把软件编写得能够方便地添加新功能，在设计阶段就永远止步不前了。

我们没有这样做的预算。

客户正在死死盯着我们的进度，要求立即实现呢。我们没有时间多想。

我们以后会考虑这个问题。

看上去似乎只有两种选择。

一种是过度分析或称过度设计——我喜欢称之为“分析瘫痪”（paralysis by analysis）。

一种是一上来就扎进细节中，编写代码，根本不考虑长期问题，然后在这种短视行为还没有引起太多问题之前，就投入下一个项目。我喜欢称之为“放任自流”[或者“自暴自弃”，abandon (by) ship (date)]。

因为管理层所受的压力是交付产品而不是维护，因此出现这样的结果也许并不令人感到惊讶。但是，稍做反思吧。是否可能有第三种选择呢？我们的基本假设和信条是否阻碍了我们看到其他选择呢？这种情况下，对于需要考虑变化的设计来说，固有信条所带来的代价比无需担心变化的设计要高昂得多。

然而，这种信条往往是错误的。事实常常恰好相反：回头再考虑系

统会如何随时间而发生变化，往往会发现更好的设计。这比以前视为标准的那种“匆匆忙忙先解决问题”的方式所花费的时间经常更少，而且代码的质量也会更高——更易于阅读、测试和修改。这些因素足以弥补正确行事可能花费的更多时间。

考虑变化的设计

下面的案例研究很好地说明了“考虑变化的设计”方式。请注意我并不是要准确预测出变化性质如何。相反，我假设变化将会出现，并尝试预测其出现的位置。在《设计模式》一书中这样描述此方式所基于的原则。

“针对接口进行编程，而不要针对实现编程。”[25]

“优先使用对象组合，而不是类继承。”[26]

“考虑设计中什么应该是可变的。这种方法与关注引起重新设计的原因刚好相反。它不是考虑什么会迫使设计发生改变，而是考虑什么能够在不引起重新设计的前提下改变。这时主要关注的就是对变化的概念进行封装，这是许多设计模式的主题。”[27]

我的建议是：需要修改代码处理新的需求时，至少应该对这些策略予以考虑。如果遵循这些策略不会给设计和实现带来明显的开销增加，就应该照做。这样做可以获得长期的好处，而且短期的开销（即使存在）并不大。

但是，我并不赞成盲目套用这些策略。候选设计方案的价值可以通过面向对象设计原则来检验。这种方法本质上与在我第10章中推演Bridge模式时所用的方法相同。在第10章中，通过查看候选设计符合面向对象原则的程度，来测度其质量。

9.3 国际电子商务系统案例研究：最初的需求

电子商务：一个关于需求的案例研究

在这个新的案例研究中，我们来考虑一个美国某国际电子商务公司的订单处理系统。这个系统必须能够处理许多不同国家（地区）的订单。本章中，需要考虑需求变化的挑战，还有应对这种情况的各种方法。在第16章中，我们继续讨论这个案例研究，届时将主要关注可变性问题。

这个系统的总架构中有一个控制器对象，用于处理销售请求。它能够确认何时有人在请求销售订单，并将请求转发给SalesOrder对象进行订单处理。

系统大致如图9-1所示。

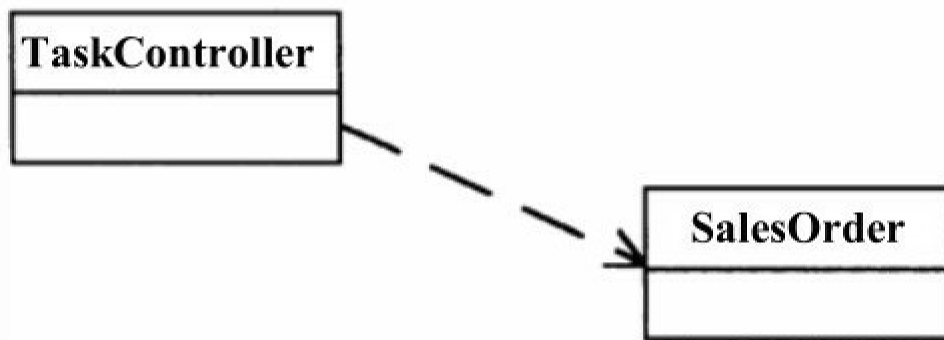


图9-1 某电子商务系统的订单体系结构

SalesOrder对象的功能包括：

系统的一些功能

允许客户通过GUI填写订单；

处理税额的计算；

处理订单，打印销售收据。

有些功能可能需要借助其他对象实现。比如，SalesOrder对象没有必要自己打印，它的作用是充当一个存放销售订单信息的容器。具体的SalesOrder对象可以调用SalesTicket对象来打印。

9.4 处理新的需求

缴税规则的新需求

在这个应用程序编写完成之后，假设我收到一个新的需求，要修改处理税额的办法。比如，我现在必须处理美国之外的顾客的订单税额。我至少需要添加新的税额计算规则。

都有哪些办法呢？新规则应该怎样处理？

在考虑这个具体情况之前，我们先说几句题外话。一般来说，应该怎样处理概念上完全相同的任务的不同实现呢（比如处理不同的税额计算规则）？我的脑中立即浮现出以下的几种候选方案：

复制和粘贴；

使用switch或者if语句，用一个变量指定各种情况；

使用函数指针或者委托（让另一个代表每一种情况）；

继承（让派生类用新的方式处理）；

将整个功能委托给新的对象。

复制和粘贴

还是老办法。我已经有了一些能够工作的代码，现在需要的与之非常相近。因此，只需要复制并且粘贴，然后对其进行修改。当然，这会给维护带来麻烦，因为现在我或者其他人必须维护同一代码的两个版本。我忽略了重用对象的可能性，就整个公司而言，最后的维护成本肯定更高了。

分支语句

这是一个看似合理的方法，但是对于需要随时间不断演进的应用程序而言，这种方法存在一些严重问题。想想在几个分支语句中使用一个变量对耦合性和可测试性的影响吧。例如，假定我使用局部变量myNation表示客户所在国家（地区）。如果选项只有美国和加拿大，那么使用分支语句可能没有问题。比如可能会有这样的分支语句：

```
// 税额 switch (myNation) {
   case US:
    // 美国计税规则 break;
   case Canada:
    // 加拿大计税规则 break;
}
```

```
// 货币 switch (myNation) {
   case US:
    // 美国货币规则 break;
   case Canada:
    // 加拿大货币规则 break;
}
```

```
// 日期格式 switch (myNation) {
   case US:
    // 使用 mm/dd/yy 格式 break;
   case Canada:
    // 使用 dd/mm/yy 格式 break;
}
```

可是如果选项更多会怎么样呢？例如，假设我需要在国家（地区）列表中添加德国，从而还要增加语言。现在代码将如下所示。

```
// 税额 switch (myNation) {
   case US:
    // 美国计税规则 break;
   case Canada:
    // 加拿大计税规则 break;
   case Germany:
    // 德国计税规则 break;
}
```

```
// 货币 switch (myNation) {
   case US:
    // 美国货币规则 break;
   case Canada:
    // 加拿大货币规则 break;
   case Germany:
    // 欧洲货币规则 break;
}
```

```
// 日期格式 switch (myNation) {
   case US:
    // 使用 mm/dd/yy 格式 break;
   case Canada:
   case Germany:
    // 使用 dd/mm/yy 格式 break;
}
```

```
// 语言 switch (myNation) {
   case US:
   case Canada:
    // 英语 break;
   case Germany:
    // 德语 break;
}
```

这还不算太坏，但是请注意分支语句已经没有之前那么从容了。现在各个分支里还是直通的，可我终究还是有可能需要在分支内部添加选项。突然，事情变得很糟。例如，添加加拿大魁北克的法语之后，代码变成了：

```
// 语言 switch (myNation) {
   case Canada:
      if ( inQuebec) {
         // 法语 break;
      }
   case US:
      // 英语 break;
   case Germany:
     // 德语 break;
}
```

分支本身的流向也开始模糊了。难以阅读，难以理解。每次增加新的分支时，程序员都必须搜遍各个角落，找出可能涉及的地方（往往会遗漏一个）。我喜欢称之为“分支蔓延”。

函数指针和委托

C++中的函数指针和 C#中的委托都可以用来将代码隐藏在精巧、紧凑、内聚的函数之中。但是，函数指针和委托无法维持每个对象的状态，因此其使用也是受限的。

继承

这是一种新方法。继承经常被人误用，这使它的名声不佳。其实继承本身并没有什么问题（对它的误解我深表遗憾）。但是如果错误使用，继承会使设计非常脆弱和僵化。这种误用的根本原因在于面向对象原则教学中存在问题。

当面向对象设计成为主流时，“重用”曾经被吹捧为它的主要优点之一。为了实现“重用”，教学中总是强调应该找到已有的东西，用派生类的形式对其进行小幅修改。[28]

在我们的缴税实例中，可以试图重用现有的SalesOrder对象。我可以将新缴税规则看成新种类的销售订单，只是缴税规则不同。例如，对于加拿大销售订单，可以从SalesOrder派生名为CanadianSalesOrder的新

类，改写缴税规则。这一解决方案如图9-2所示。

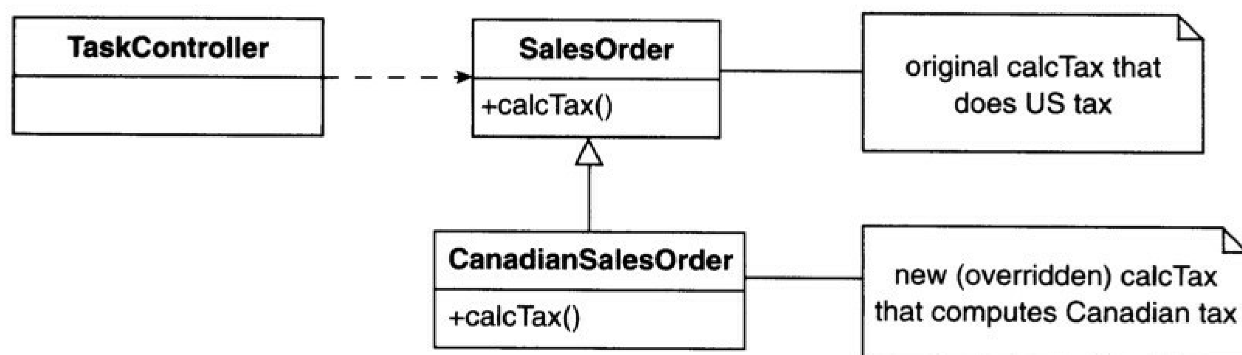


图9-2 电子商务系统的销售订单架构

……但是这会产生问题

采用这种方法的困难之处在于，它这次能够奏效，但是无法次次奏效。比如，如果要处理德国订单，或者应对其他方面发生变化（如日期格式、语言和运费规则），我们构建的继承层次将无法轻松地应对。诸如此类的反复特化，要么会使代码变得无法理解，要么产生冗余。人们对面向对象设计经常有一种抱怨：特化技术最终总是会产生太深的继承层次。糟糕的是，继承层次太深将导致程序难以理解（弱内聚）、存在冗余、难以测试而且多个概念耦合在一起。无怪乎许多人认为面向对象有些言过其实——尤其是这一切都是因为遵循了通用的面向对象“重用”要求。

设计模式采取的是另一种方法

我能采用什么不同的办法呢？应该遵循前面所述的规则，尝试“考虑设计中什么应该是可变的”、“对变化的概念进行封装”，并且最重要的是“优先使用对象聚集，而不是类继承”。[29]

根据这种方法，应该这样做：

1.寻找变化，并将它封装在一个单独的类中；

2.将这个类包含在另一个类中。

第1步：发现变化并封装之

在本例中，已经确定缴税规则是变化的。“将它封装”就意味着创建一个抽象类定义如何在概念上完成税额计算，然后为每种变化派生具体类。也就是说，可以创建一个 CalcTax 对象，为完成税额计算这一任务定义接口。然后可以由它派生所需的特定版本，如图9-3所示。

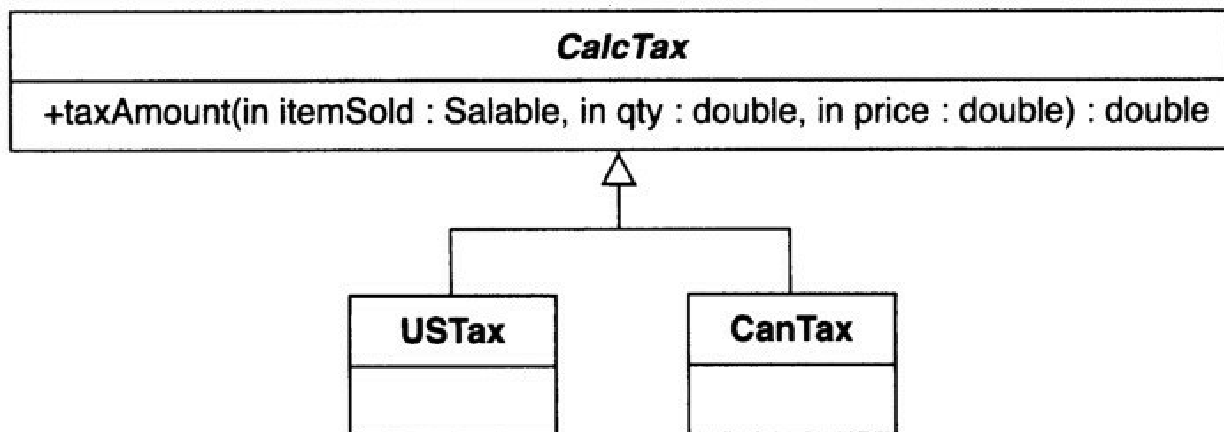


图9-3 封装缴税规则

第2步：组合优先

接下来，应该使用组合取代继承。这意味着用不着再创建不同版本的销售订单（使用继承），可以用组合来包含变化。也就是说，只有一个SalesOrder类，让它包含处理变化的CalcTax类，如图9-4所示。

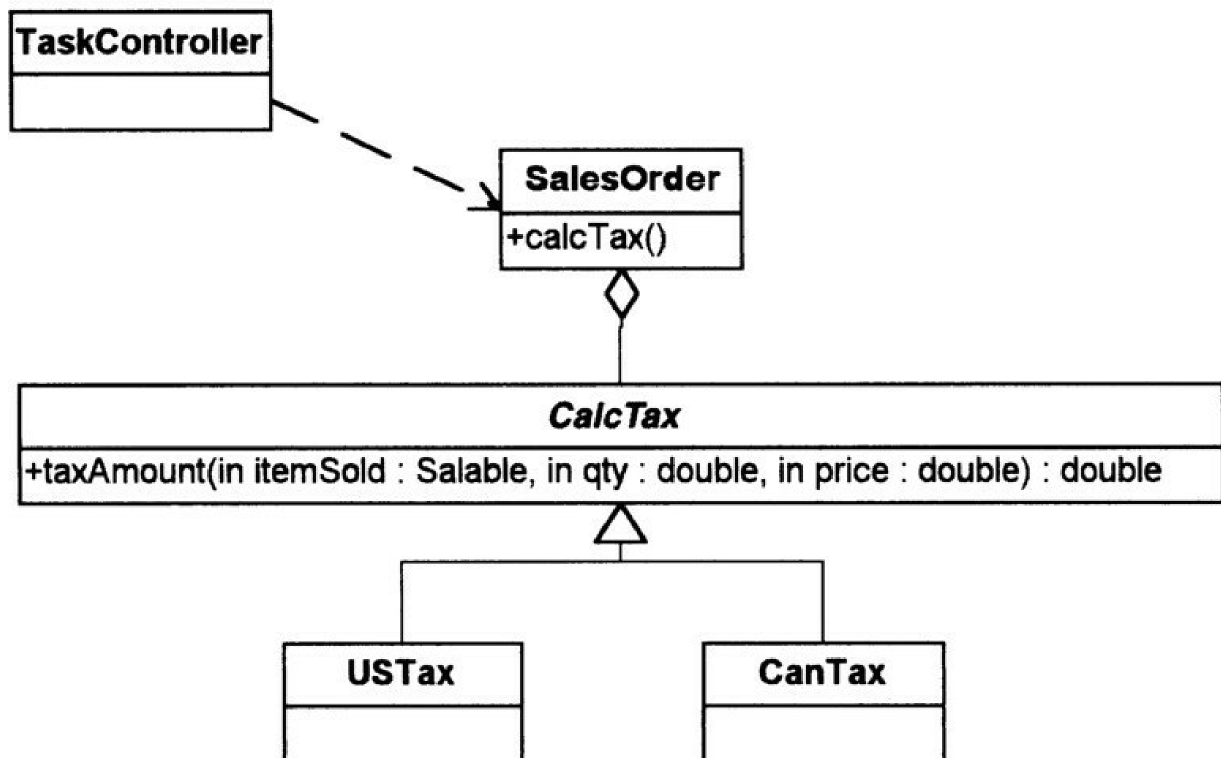


图9-4 用组合代替继承

例**9-1 Java**代码片段：实现**Strategy**模式

```
public class TaskController {
   public void process () {
      // 这里的代码模拟了处理任务控制器
      // ...
      // 取得所在国信息
      CalcTax myTax;
      myTax= getTaxRulesForCountry();
      SalesOrder mySO= new SalesOrder();
      mySO.process( myTax);
   }
   private CalcTax getTaxRulesForCountry() {
```

```
        // 在实际开发中，要取得所在国的计税规则。
        // 相应逻辑可以写在这里，也可以放到配置文件中。
        // 这里返回USTax就可以编译了。
        return new USTax();
    }
}
public class SalesOrder {
    public void process (CalcTax taxToUse) {
        long itemNumber= 0;
        double price= 0;
        // 创建计税对象
    // ...
    // 计算税额
    double tax=
        taxToUse.taxAmount( itemNumber, price);
    }
}
public abstract class CalcTax {
    abstract public double taxAmount(
        long itemSold, double price);
}
public class CanTax extends CalcTax {
    public double taxAmount(
        long itemSold, double price) {
        // 在实际开发中，要根据加拿大的计税规则来编写代码。
        // 在此返回0即可编译。
        return 0.0;
```

```
    }
}
public class USTax extends CalcTax {
    public double taxAmount(
        long itemSold, double price) {
        // 在实际开发中，要根据美国的计税规则来编写代码。
        // 在此返回0即可编译。
        return 0.0;
    }
}
```

UML图

UML 中可以在方法里定义参数。这是通过在方法名之后的括号中给出一个参数及其类型来实现的。

因此，在图9-4中，taxAmount方法有三个参数：

Salable类型的itemSold

double类型的qty

double类型的price

所有这些参数都是用“in”标记的输入参数。taxAmount方法还有一个double返回值。

工作原理

我为 CalcTax 对象定义了一个非常通用的接口。显然，应该有一个Saleable类定义可以销售的货物（及其缴税方式）。SalesOrder对象将Saleable对象与数量和价格一起提供给CalcTax对象。CalcTax对象需要的所有信息就齐了。

提高内聚度，有助于灵活性

这种方法的另一个优点在于提高了内聚度。销售税有专门的类进行处理。还有一个优点是：在有新的缴税需求时，只需从 CalcTax 类派生一个新类予以实现即可。

最后，这种方法使职责的转移更加容易了。例如，在基于继承的方法中，必须由TaskController决定该使用哪个类型的SalesOrder对象。而在新结构中，SalesOrder对象的类型既可以由TaskController对象来决定，也可以由SalesOrder对象决定。为了由SalesOrder对象决定，需要有一个Configuration（配置）对象，使SalesOrder对象知道应该使用哪个税额计算对象（可能就是 TaskController对象使用的对象），如图9-5所示。

使职责的转移更容易

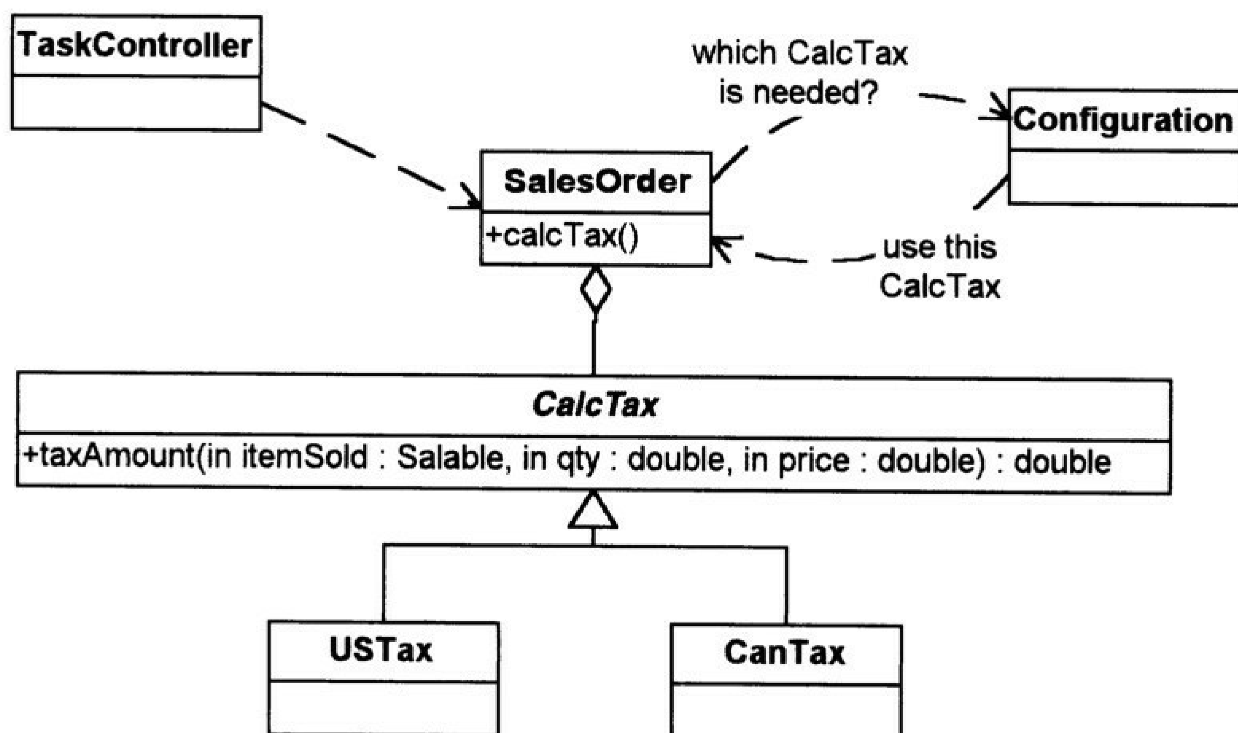


图9-5 使用Configuration告诉SalesOrder对象应该使用哪一个CalcTax

与直接继承的比较

大多数人都会注意到这种方法也使用了继承。确实如此。但是，它

使用继承的方式与仅仅从 SalesOrder派生 CanadianSalesOrder是不同的。在严格的继承方法中，是通过在SalesOrder继承来处理变化的。而在设计模式所指导的方法中，使用了对象聚集。（也就是说，SalesOrder中包含一个引用，指向处理变化的功能也就是税额计算的对象。）从SalesOrder（需要扩展的类）的角度来看，是用组合代替了继承。至于被包含的类如何处理变化，SalesOrder并不关心。

有人可能会问了：“可是，你这不就是将问题向下推了吗？”对这个问题要从三方面回答。首先，这种说法没错，但是这样做能够简化更大、更复杂的程序。其次，原设计在一个类层次（SalesOrder）中装入了许多独立的变量（税额、日期格式等等），而新的方法将这些变量都放在自己的类层次中，这样就能够独立地分别扩展它们。最后，在新方法中，系统的其他部分可以独立于SalesOrder使用（或者测试）这些更小的操作。总而言之，模式所提倡的方法伸缩性更强，这是原来直接使用继承的方法所不具备的。

处理新情况和规范化

将某个变化的行为从使用它的类中移出来，这种过程与数据库中的规范化过程非常相似，在后一种过程中，需要将列移到自己的表中，使用外键引用它们。

Strategy模式

这种方法使业务规则能够独立于使用自己的SalesOrder对象而发生改变。请注意对于目前的变化和未来可能出现的任何情况，这种方法都行之有效。这种“将算法封装在一个抽象类（CalcTax）中，而且在某一时刻能够互换地使用其中之一”的方法，本质上就是Strategy模式。

9.5 Strategy模式

意图，来自《设计模式》一书

《设计模式》一书中对Strategy模式的意图是这样叙述的：

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。Strategy模式使算法可独立于使用它的客户而变化。[30]

Strategy模式的动机

Strategy模式以下列几条原则为基础：

对象都具有职责。

这些职责不同的具体实现是通过多态的使用完成的。

概念上相同的算法具有多个不同的实现，需要进行管理。

将问题域中的各个行为互相分离开来——也就是说将它们解耦，是一个好的设计实践。这使我们可以修改负责某一行为的类，不会对其他类产生不良影响。

Strategy模式：关键特征

意图

可以根据所处上下文，使用不同的业务规则或算法。

问题

对所需算法的选择取决于发出请求的客户或者要处理的数据。如果只有一些不会变化的算法，就不需要Strategy模式。

解决方案

将对算法的选择和算法的实现相分离。允许根据上下文进行选择。

参与者与协作者

Strategy指定了如何使用不同的算法。

各ConcreteStrategy实现了这些不同的算法。

Context通过类型为Strategy的引用使用具体的Concrete-Strategy。

Strategy与Context相互作用以实现所选的算法（有时候Strategy必须查询Context）。Context将来自Client的请求转发给Strategy。

效果

Strategy模式定义了一系列的算法。

可以不使用switch语句或条件语句。

必须以相同的方式调用所有的算法（它们必须拥有相同的接口）。各 ConcreteStrategy 与 Context 之间的相互作用可能需要在Context中加入获取状态的方法。

实现

让使用算法的类（Context）包含一个抽象类（Strategy），该抽 象类有一个抽象方法指定如何调用算法。每个派生类按需要实现算法。注意，在原型Strategy模式中，选择所用具体实现的职责由Client对象承担，并转给Strategy模式的Context对象。

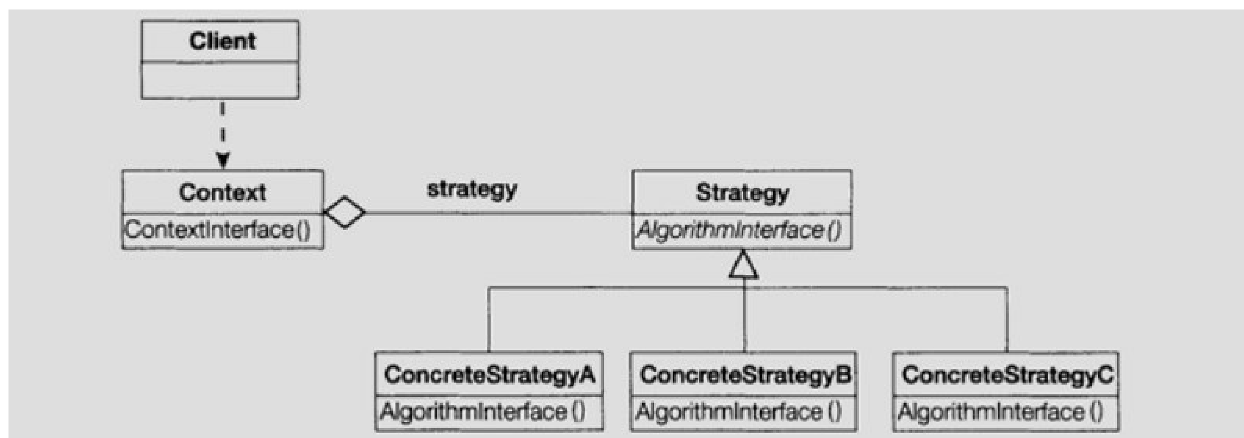


图9-6 Strategy模式的通用结构图

9.6 实践注记：使用Strategy模式

这些限制检验了Strategy模式

我在模式课程中使用过这个电子商务示例，曾经有人问道："你知不知道在英国超过一定年龄的人不需要为食物消费上税？"我对此一无

所知，CalcTax对象的接口无法应对这种情况。但是，我可以至少用3种方法予以解决。

1.将 Customer对象的年龄成员传给 CalcTax对象，如果需要就使用它。

2.更具一般性的做法是，将 Customer对象本身传递给 CalcTax对象，如果需要就对其进行查询。

3.再更具一般性的做法是，传递一个指向 SalesOrder 对象的引用（即this引用），让CalcTax对象查询它。

虽然为了应对这种情况，还是必须修改SalesOrder类和CalcTax类，但修改方式很明显。而且，不大可能引入新问题。

封装业务规则

从技术角度而言，Strategy模式就是用来封装算法的。但是在实践中，我发现可以用它来封装几乎任何类型的规则。一般来说，只要在分析过程中听到需要在不同时间应用不同业务规则，我就会考虑使用Strategy模式处理这种变化的可能性。

Context和Strategy之间的耦合

Strategy 模式要求所封装的算法（业务规则）应处在使用它们的类（Context）之外。这意味着Strategy所需信息必须要么传递给它们，要么以某种其他形式获得。

降低单元测试成本

Strategy 模式还简化了单元测试，因为每个算法都有自己的类，可以通过自己的接口单独测试。如果算法不像 Strategy 模式中那样移出来，Context 和 Strategy 之间的耦合将使测试非常困难。例如，在实例化Context对象之前可能有些前提条件。或者，Context可能通过保护数据成员提供Strategy所需信息。如果同时存在几个不同系列的算法，测试能够进一步简化。（也就是说，存在几个 Strategy 模式，这种情况往往会出现。）这是因为使用Strategy模式，开发人员不需要操心与

Context耦合所带来的各种相互作用。也就是说，我们应该能够独立地测试每个算法，而无需担心可能的所有组合情况。

当总是使用相同的Strategy对象时

在前面的销售订单示例中，每次需要时，TaskController就将策略对象传递给SalesOrder对象。稍加反思就会发现，除非要为不同的客户重用销售订单，我应该总是对任何特定的SalesOrder对象使用相同的策略对象。我经常使用的一种Strategy模式变体是，在Context后构造函数中将策略对象赋予 Strategy 模式（在本例中是 SalesOrder 对象）中的Context。然后任何需要引用它的方法都可以使用了，无需再要求传入。但是，由于Context仍然不知道Strategy对象的具体类型，模式的威力依旧。如果在Context对象构造时知道需要哪个Strategy对象，就可以应用这一变体。

对Strategy模式所带来的类爆炸性增长问题的各种消除方法

有时候，有些学生抱怨Strategy模式会增加大量的类。虽然我并不认为这真的是什么问题，但是在能够控制所有策略时，可以采取一些措施尽量减少类的数量。在这种情况下，如果使用的是C++，可以用Strategy抽象类的头文件包含所有 ConcreteStrategy 类的头文件。同时用Strategy抽象类的cpp文件包含各ConcreteStrategy的代码。如果使用的是Java，可以在 Strategy 抽象类内使用内部类包含所有的ConcreteStrategy。但是如果无法控制所有的 Strategy对象——也就是说如果有其他程序员需要实现自己的算法，就不要这样做。

9.7 小结

本章内容

Strategy 模式是一种定义一系列算法的方法。概念上来看，所有这些算法完成的都是相同的工作，只是实现不同。

本章给出了一个例子，使用一系列税额计算算法。在一个国际电子商务系统中，不同的国家（地区）需要使用不同的税额计算算法。通过Strategy模式，可以将这些规则封装在一个抽象类中，然后派生出一系列的具体类。

通过从一个抽象类派生执行算法的所有不同方式，主模块（上例中的SalesOrder对象）就无需再操心实际使用的是哪一个。这样能够允许发生新的变化，但是又增加了一个需求：管理这些变化——这是我们将在第16章中讨论的一个问题。

复习题

简答题

1.处理新的需求有哪些方案？

2.《设计模式》一书中提出的指导如何应对变化的三项基本原则是什么？

3.Strategy模式的意图是什么？

4.Strategy模式的效果有哪些？

阐述题

1.《设计模式》建议“考虑设计中什么应该是可变的。”这与关注引起重新设计的原因有何不同？

2.复制和粘贴的方式有什么问题？

3.“分支蔓延”是指什么？

4.处理变化的设计模式方法有何优势？

5.为什么对象聚集的继承方式在应对变化时优于直接类继承？

观点与应用题

1.你曾经遇到过什么情况下，无法负担出现变化？为什么会出现这

种情况，结果又如何呢？

2.应该使用分支语句吗？为什么？

第10章 Bridge模式

10.1 概览

本章内容

本章我们通过Bridge（桥接）模式继续设计模式的学习。Bridge模式比我们前面已经学过的模式要复杂一些。当然它也有用得多。

在本章中，我们将：

提供一个例子，从中推出 Bridge 模式。我们将比较深入地讨论细节，帮助你学习这个模式；

给出Bridge模式的关键特征；

阐述一些来自我自己经验的有关Bridge模式的几点知识。

10.2 Bridge模式简介

意图：将抽象与实现解耦

《设计模式》一书中对Bridge模式的意图是这样叙述的："将抽象与其实现解耦，使它们都可以独立地变化"。[31]

难以理解

我还清楚地记得自己第一次读到这句话时的感觉：嗯？

稍后，我虽然可以理解这句话的每个字，却对整个句子的意思毫无头绪，这是怎么回事呢？

我知道：

解耦（decouple）是指让各种事物互相独立地行事，或者至少明确地声明之间的关系；

抽象（abstraction）是指不同事物之间概念上的联系方式。

而且我当时认为实现就是实际构建抽象的方式。令我不解的是，怎样才能将抽象与其实现的具体方式分离开呢？

其实我的迷惑主要是因为误解了实现的含意。这里实现指的是抽象类及其派生类用来实现自己的对象（而不是抽象类的派生类，这些派生类被称为具体类）。老实讲，就算当时对实现的理解无误，我也无法确定这能对我有多大帮助。这句话所表达的概念确实很难一下子就弄明白。

如果现在你还对Bridge模式摸不着头脑，别担心。只要理解了它的意图，就已经前进一大步了。

它确实是一个对学习者而言极富挑战性的模式，它功能非常强大。

Bridge 模式是最难理解的模式，部分原因是它功能非常强大，适用于很多场合。而且，它还与常见的用继承来处理特殊情况的方式背道而驰。但是，它却是一个遵循设计模式社区两大原则的极好例子：“找出变化并封装之”和“优先使用对象聚集，而不是类继承”。下面你将看到这一点。

10.3 学习Bridge模式：示例

了解其存在价值，然后推出模式

为了帮助读者理解Bridge模式背后的思想及其作用，我将从头开始完成一个例子。我将从需求开始，推出这个模式，然后说明如何运用它。

这个例子看起来可能比较简单。但是通过了解例子中所讨论的概念，然后思考曾经遇到的类似情况，也就是说具有如下特点：

概念的抽象有变化；

这些概念的实现方式有变化。

你就会发现这个例子与前面讨论的CAD/CAM问题有很多相似性。但是我不会一上来就先给出所有需求，我将逐步给出，一次一部分，就像实际情形中提出需求那样。在研究问题刚开始，是无法总能看到变化的。

从一个绘制形状的简单问题开始

要点：在需求定义期间，应该尽早而且经常地考虑变化!

假设我接受了一个任务：编写一个程序，使用两个绘图程序之一绘制矩形。我被告知，实例化矩形的时候，它会知道应该使用绘图程序1（DP1）还是绘图程序2（DP2）。

其中矩形是用两对点来定义的，如图10-l所示。表10-1总结了两个绘图程序之间的区别。

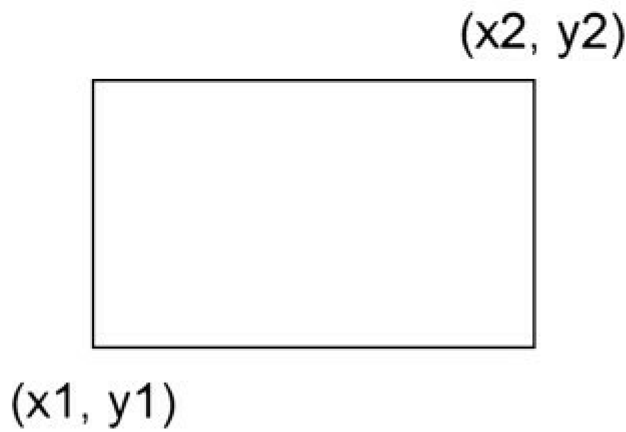


图10-1 给出矩形的位置

表10-1 绘图程序的区别

	DP1	DP2
用于画线	draw_a_line（x1，y1，x2，y2）	drawline（x1，x2，y1，y2）
用于画圆	draw_a_circle（x，y，r）	drawcircle（x，y，r）

继承的正确使用

分析师做出规定，绘制矩形的代码不需要操心自己应该使用哪种绘制程序。我想到，因为矩形在实例化的时候会知道使用哪个绘图程序，所以可以有两种不同的矩形对象：一种使用DP1，一种使用DP2。每种

矩形对象都有一个绘制方法，但实现的方式不同，如图10-2所示。

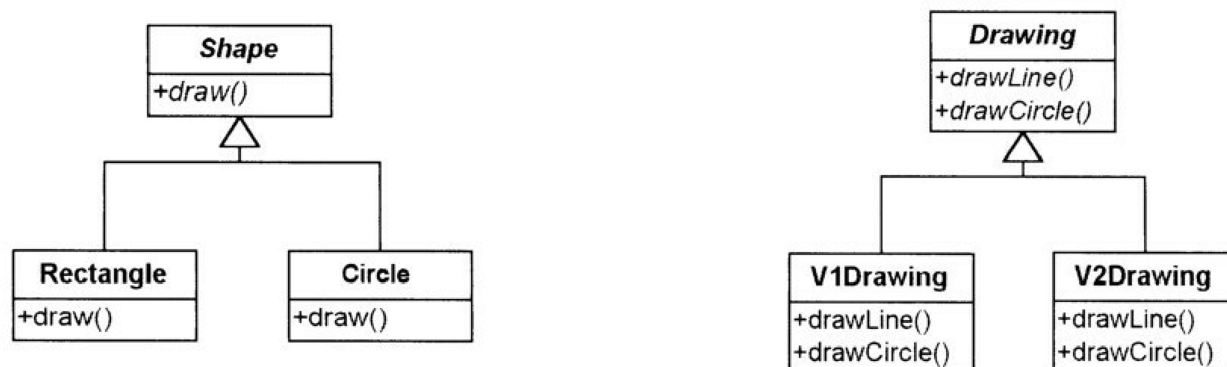


图10-2 矩形和绘图程序（DP1和DP2）的设计

关于实现的一点说明

通过引入一个抽象类　Rectangle，我可以利用这样一个事实，不同Rectangle之间唯一的差异在于如何实现drawLine方法。V1Rectangle类是通过一个DP1对象的引用，使用该DP1对象的draw_a_line方法实现的。V2Rectangle类是通过一个DP2对象的引用，使用该DP2对象的drawline方法实现的。无论怎样，通过实例化正确的Rectangle，我不必再操心这种差异了。

例**10-1 Java**代码片段

```
abstract public class Rectangle {
  private double _x1, _y1, _x2, _y2;
  public Rectangle
    (double x1, double y1, double x2, double y2) {
    x1= x1; _y1= y1; _x2= x2; _y2= y2;
  }
  public void draw() {
    drawLine( _x1, _y1, _x2, _y1);
    drawLine( _x2, _y1, _x2, _y2);
```

```
            drawLine( _x2, _y2, _x1, _y2);
            drawLine( _x1, _y2, _x1, _y1);
        }
        abstract protected void drawLine
            (double x1, double y1, double x2, double y2);
    }
```

但是，需求总是会变化的

现在，假设在代码完成之后，需求发生了变化——它和死亡、纳税都是不可避免的三件事。我现在需要支持另一种形状——这次是圆形。但是，需求还要求集合对象无需知道Rectangle和Circle的差异。

……实现还是可以很简单

我想到可以在类层次中增加一层，就可以在已经使用的方法之上进行扩展了。我只需要增加一个名为 Shape的新类，并从中派生 Rectangle类和 Circle类。这样，Client对象可以只引用 Shape对象，而不必考虑所给的是哪种Shape类。

使用继承的设计

对一名初级面向对象分析师而言，只用继承实现这些需求似乎很自然。例如，我可以从图10-2这样的设计开始，然后对每一种Shape类，都用各自的绘图程序实现，为Rectangle类派生一个DP1版本和一个DP2版本，为 Circle类也派生一个DP1版本和一个DP2版本。最终得到的设计如图10-3所示。

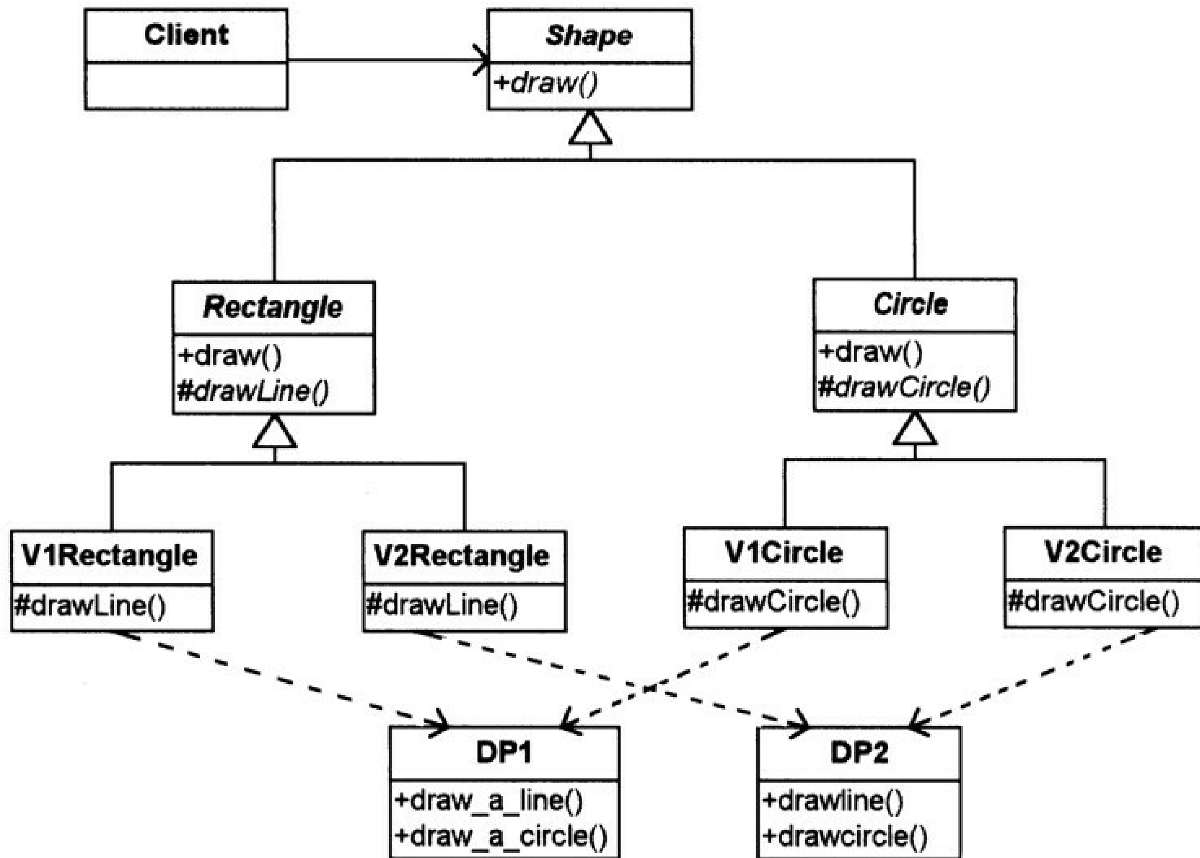


图10-3 一种直截了当的方法：实现两种形状和两个绘图程序

实现Rectangle类的方法与实现Circle类的方式相同。但是，这一次实现draw方法使用的是drawCircle方法而不是drawLine方法。

例**10-2 Java**代码片段

```
abstract class Shape {
   abstract public void draw();
}
// Rectangle 类只改变了一个地方
abstract class Rectangle extends Shape {
//
// V1Rectangle 和 V2Rectangle 不变
```

```
abstract public class Circle extends Shape {
   protected double _x, _y, _r;
   public Circle (double x, double y, double r) {
      _x= x; _y= y; _r= r;
   }
   public void draw() {
      drawCircle();
   }
   abstract protected void drawCircle();
}
public class V1Circle extends Circle {
   public V1Circle
      (double x, double y, double r) {
      super(x,y,r);
   }
   protected void drawCircle () {
      DP1.draw_a_circle( _x, _y, _r);
   }
}
public class V2Circle extends Circle {
   public V2Circle(double x, double y, double r) {
      super( x, y, r);
   }
   protected void drawCircle () {
      DP2.drawCircle( _x, _y, _r);
   }
}
```

理解设计

为了理解这个设计，让我们再看一个例子。请考虑 V1Rectangle类的draw方法的作用。

Rectangle类的draw方法和前面一样（按需要4次调用drawLine方法）。

通过调用DP1的draw_a_line方法实现drawLine方法。运行中这个设计如图10-4所示。

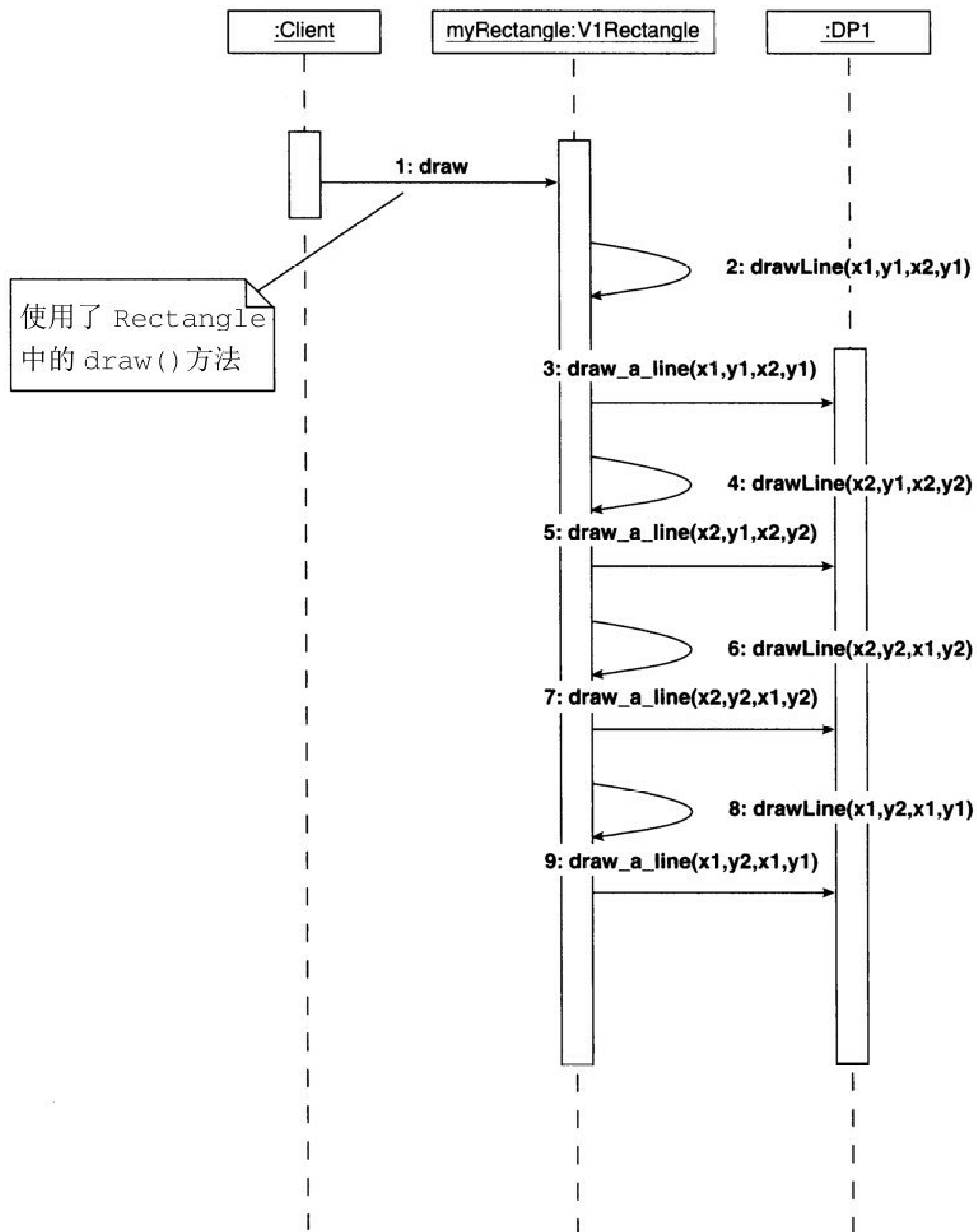


图10-4 有V1Rectangle时的顺序图

阅读顺序图

正如第2章中讨论过的，图10-4所示是一种特殊的交互图，名叫顺序图（sequence diagram）。这种图在UML中很常见，通常用于说明系统中对象之间的交互。

最上面的每个方框都表示一个对象，可能有名字，可能没有名字。

如果对象有名字，名字在冒号的左边给出。

对象所属的类在冒号的右边，因此，中间方框中的对象名为myRectangle，它是V1Rectangle类的一个实例。

读图时应该从上往下，每个编了号的语句都是一条对象发送给自己或另一对象的消息。

整个序列开始是无名的Client对象调用myRectangle对象的draw方法。

draw方法调用myRectangle对象自己的drawline方法4次（如第2、4、6和第8步所示）。请注意时间线上的箭头指回myRectangle对象。

drawLine调用DP1的draw_a_line方法，如第3、5、7和第9步所示。

虽然类图中似乎有很多对象，但实际上，只需处理 3个对象（如图10-5所示）：

使用矩形的Client对象；

V1Rectangle对象；

绘图程序DP1对象。

当 Client对象向 V1Rectangle对象（名为 myRectangle）发送消息，要求它执行 draw方法时，V1Rectangle对象将调用 Rectangle的draw方法，引发了图10-4中第2～9步。



图10-5 顺序图中出现的对象

这个解决方案存在“组合爆炸”问题

糟糕的是，这个办法带来了新的问题。请看图10-3，注意第三行中的类。考虑以下几种情形。

这一行的类表示的是Shape的4个具体类型。

如果我还有另一个绘图程序，也就是说实现上又有一种新变化，会怎么样呢？将会有6种不同类型的Shape（2个概念上的Shape乘以3个绘图程序）。

想象一下，如果我还有另一类型的Shape，也就是说另一种概念上的变化，会怎么样呢？将会有9种不同类型的Shape（3个概念上的Shape乘以3个绘图程序）。

……因为是紧耦合的

于是类爆炸性增长的问题出现了，因为这个解决方案中抽象（Shape的种类）与其实现（绘图程序）是紧耦合的。每种形状都必须知道自己用的是哪种绘图程序。需要有一种方式将抽象上的变化和实现上的变化分开，从而使类的数量仅仅是线性地增加（见图10-6）。



图10-6 Bridge模式能够将抽象上和实现上的变化分开

这正是Bridge模式的意图：将抽象与其实现解耦，使它们都可以独立地变化。[32]

我们的不良设计带来的另外几个问题

在给出解决方案并推出Bridge模式之前，我要先提一下另外几个问题（除了组合爆炸之外）。

再看图10-3，问问自己这个设计还有什么毛病。

看上去是否存在冗余？

是高内聚的还是低内聚的？

是紧耦合的还是松耦合的？

你希望维护根据它编写的代码吗？

过度使用继承

在我还是一个初级的面向对象分析师时，曾经很喜欢利用继承，使用特殊情况解决这里碰到的问题。我喜欢继承的思想，因为它看上去很新颖而且功能强大。只要可以用的地方我都使用继承。似乎对于许多初级分析师来说这很正常，但是其实这是很幼稚的：有了新“锤子”，所有的东西看上去都成了钉子。糟糕的是，许多教授面向对象设计的方式关注点都放在了通过特化处理变化、从已有类派生新类上。正是由于这种对对象的“is-ness”性质的过度关注，程序员往往会在巨大臃肿的类层次中创建对象，这种层次开始时可能还能正常工作，但是随着时间推移将变得越来越难以维护（我们在第9章中曾经讨论到这一点）。

而当我成为一个有经验的面向对象设计人员之后，我仍然深陷基于继承的设计方式之中，还是根据类的“is-ness”性质观察类的特点，无论结构已经变得多么复杂。

用设计模式进行思考最终救我于泥潭之中。我自此学会了用对象的职责而不是其结构来思考问题。

有经验的面向对象分析师都已经了解到应该有选择地使用继承，才能发挥其优势。使用设计模式，将有助于加快这一学习进程。其中就包括从“为每种变化使用不同的特化”（继承）到“将变化转移到使用或拥有这种变化的对象中”（组合）的转变。

一种替代方案

刚开始考虑这些问题时，我觉得目前的困难可能只是由于使用了错

误的继承层次造成的。所以，我尝试选择图10-7所示的另一种层次。

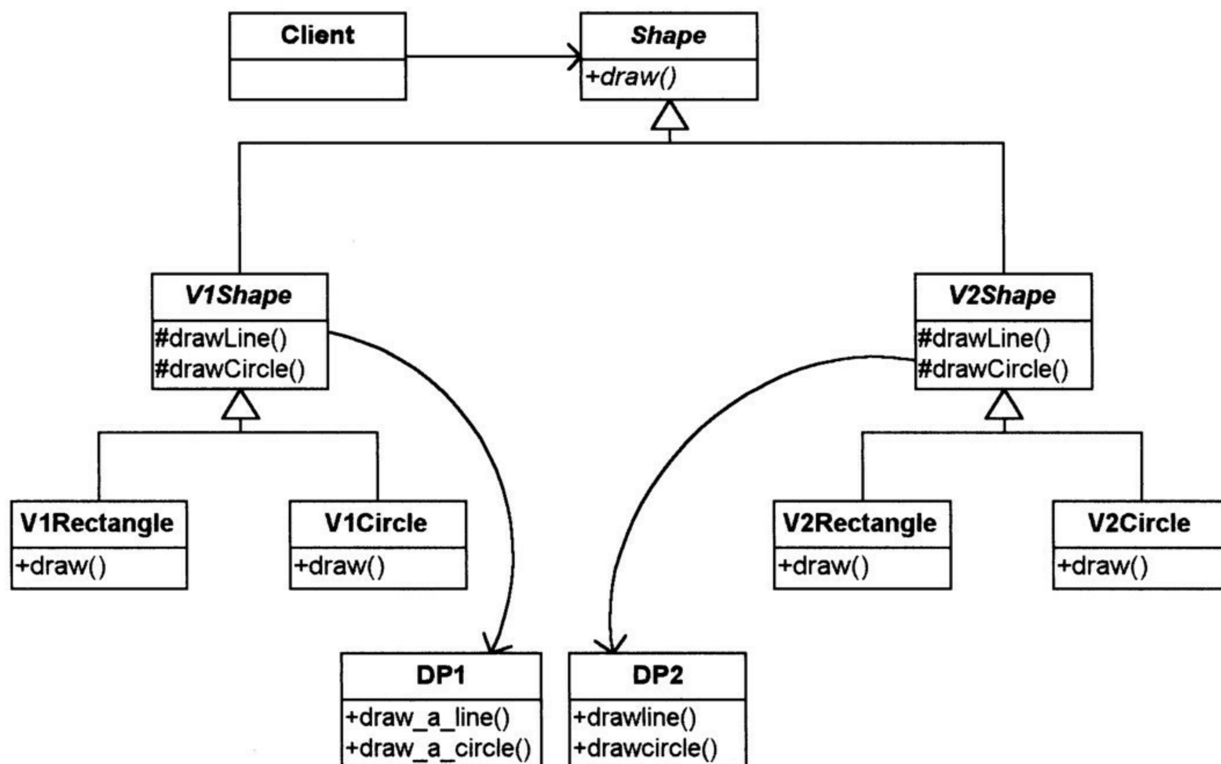


图10-7 另一种实现

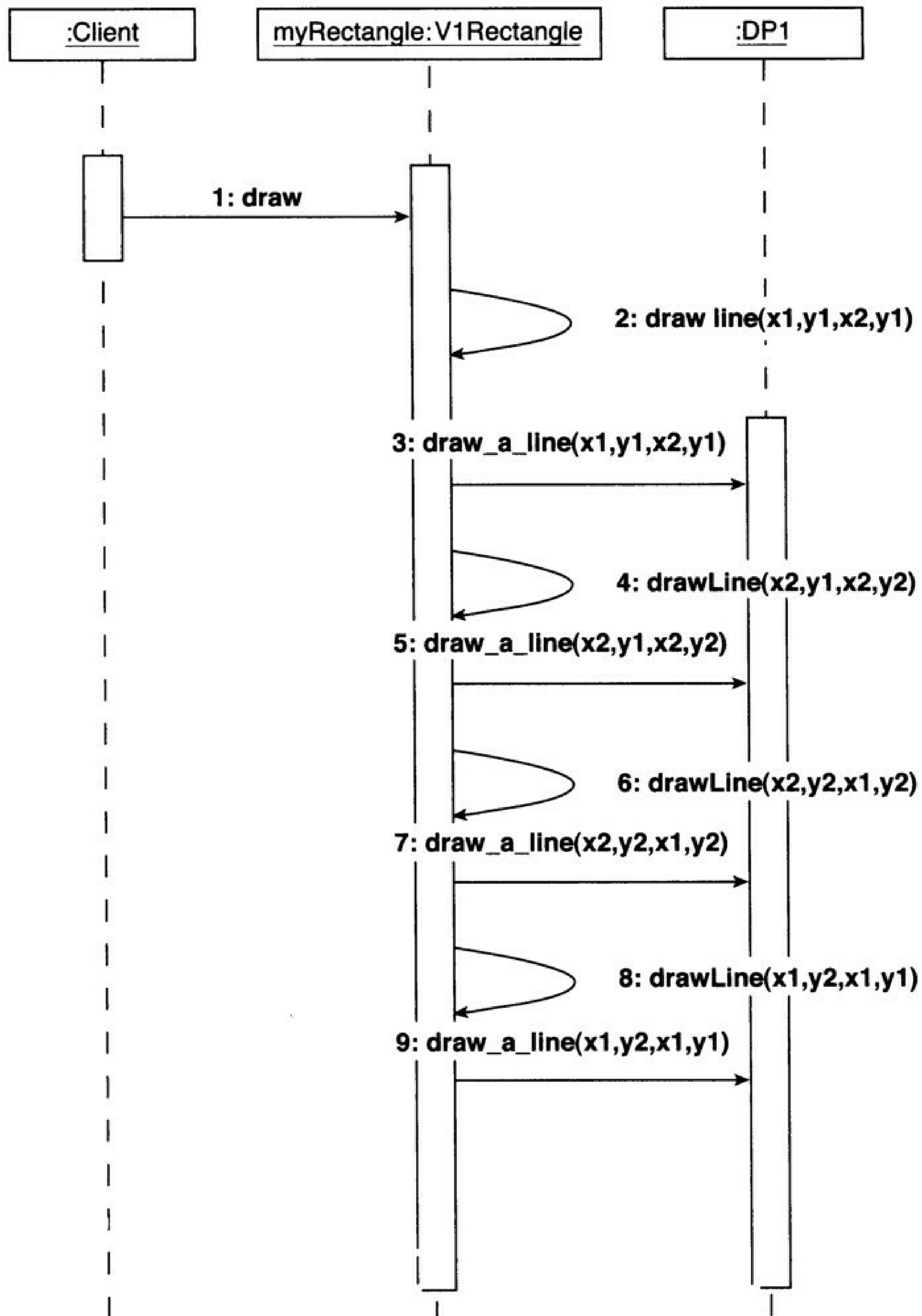
其实没有好多少，半斤八两而已

我仍然用相同的四个类表示所有可能的组合。但是，因为先从不同的绘图程序派生不同版本，所以DP1和DP2类之间的冗余去除了。

糟糕的是，两种Rectangle和两种Circle之间的冗余却没办法去除，因为它们每一对都有相同的draw方法。

无论怎样解决，前面的类爆炸性增长问题依然存在。

这一解决方案的顺序图如图10-8所示。



:Client
myRectangle:V1Rectangle
:DP1
1: draw
2: draw line(x1,y1,x2,y1)
3: draw_a_line(x1,y1,x2,y1)
4: drawLine(x2,y1,x2,y2)
5: draw_a_line(x2,y1,x2,y2)
6: drawLine(x2,y2,x1,y2)
7: draw_a_line(x2,y2,x1,y2)
8: drawLine(x1,y2,x1,y1)
9: draw_a_line(x1,y2,x1,y1)

图10-8 新方式的顺序图

仍然有可伸缩问题

尽管这种方案可能对原解决方案有所改进，但它仍然存在可伸缩问题。原来的内聚和耦合的问题也没有完全解决。

结论：我仍然不愿意维护这个版本！肯定有更好的方式。

寻找原设计的替代方案

虽然我这里给出的替代设计并不比原设计有显著的优点，但是应该指出的是，勇于寻找原设计的替代方案是一种很好的做法。太多的开发人员喜欢对最开始提出的设计方案从一而终。我并不是说要深入研究所有可能的方案（这是“分析瘫痪”的另一种方式）。但是，回头看看，考虑怎样克服原设计中的缺陷的做法，是非常值得提倡的。实际上，正是这种回头看看，拒绝继续抱着已有的、不好的设计不放的做法，促使我理解了使用设计模式的强大方法——本书整个要讲的正是这些方法。

10.4 对使用设计模式的观察

看待设计模式的新方式

当人们开始学习设计模式时，他们经常把注意力放在模式提供的解决方案上。这看起来似乎很合理，因为设计模式被广为宣传的一点就是能够为实际问题提供优秀的解决方案。

但是，这从方向上来说就是错误的。在尝试将某个解决方案应用到一个问题之前，应该先理解问题。这种只是寻找在何处应用模式的方法，只能告诉你要做什么，但是不能告诉你什么时候或者为什么使用。

我发现将注意力放在模式的上下文——模式试图解决的问题上，要有用得多。这能够使我们知道什么时候或者为什么使用模式。这与Alexander的模式理念也更相符：“每个模式都描述了一个在我们的环境中会不断重复出现的问题，并进而叙述了这个问题解决方案的要素……”[33]

本章到目前为止的讨论也证明了这一点。Bridge模式要解决的问题是什么呢？

当存在一个抽象有不同实现时Bridge模式最为有用，它可以使抽象和实现相互独立地进行变化。

此处问题的特点很好地符合这一条件。我知道我应该使用 Bridge模式，虽然我还不知道如何实现它。使抽象和实现相互独立地变化，将意味着我可以在不改变实现的情况下增加新的抽象，反之亦然。

当前的解决方案并不允许这种独立地进行变化。我已经看到，创建一个支持独立变化的实现更好。

结论：重要的是应该认识到：即使在不知道如何实现Bridge模式时，也能肯定这种情况下可以使用该模式。你会发现对于设计模式这通常都是成立的：可以在确切知道如何实现之前就确定什么时候可以在问题域中应用它们。

10.5 学习Bridge模式：通过将它推演出来

推出解决方案

完整地了解这个问题之后，我们现在可以推演出Bridge模式了。推演模式的过程将有助于更深刻地理解这个复杂而强大的模式。

我们来应用优秀面向对象设计的一些基本策略，看看它们是怎样帮助我们开发出与Bridge模式非常类似的解决方案。为此，我将使用Jim Coplien在共性和可变性分析方面的成果。[34]

设计模式就是能够反复应用的解决方案

设计模式就是能够重复应用于多个问题的解决方案，而且是经历了时间考验、已经证明其优秀的解决方案。我在本书中采取的方法，是通过推演出模式来教授模式，从而使你能够理解模式的特性。

在本章中，我已经知道要推演的是Bridge模式，因为我在《设计模式》一书中读到过这个模式，而且见过它如何在具体的问题域中发挥作用。重要的是，应该注意实际上模式并不真的是这样推演出来的。按照定义，它们必须重复出现——至少在 3个独立情形下出现过，才能考虑成为模式。这里“推演”这个词的意思是：我们将经历一个设计过程，在此过程中创造出一个之前并不知道的设计模式。这样做的目的是为了说明一些关键的原则和有用的策略。这还说明了解这些原则至少与了解模式同样重要，因为原则总是适用的，而模式只是在某些场景下才适用。

寻找对象的新范型

面向对象设计方法中，设计人员应该了解问题域，找到其中的名词，然后创建对象来表示这些名词，这些几乎是天经地义的。接下来设计人员寻找与这些名词相关的动词（也就是它们的操作），并通过在对象中添加方法来实现这些动词。这种以名词和动词为中心的过程通常会生成超出我们意愿的庞大类层次。我认为，在创建对象时使用共性和可变性分析作为主要工具，要优于仅仅关注名词和动词。（我相信这里实际上是重述了 Jim Coplien的工作。）

应对变化的策略

在进行设计以应对变化的过程中，应遵循两条基本策略：

找出变化并封装之；

优先使用对象聚集，而不是类继承。

过去，开发人员常常通过大的继承树来协调这些变化。但是，第二条策略说，尽可能地尝试使用聚集。其意图是能够在互相独立的类中包

含变化，从而允许未来发生变化，而不影响原有的代码。实现这一目的的一种方法是：将所有的变化都分别放在自己的抽象类中，然后观察这些抽象类之间的关系。

再谈封装

大多数面向对象开发人员都知道“封装”是指数据隐藏。糟糕的是，这是一个局限性很大的定义。不错，封装确实隐藏数据，但是它还可以用于很多其他方面。如果翻回去看看图7-2，会发现封装在很多层次上都发挥着作用。当然，在每个具体的 Shape 类中它起到的就是数据隐藏的作用。但是，请注意Client对象并不知道具体种类的Shape。也就是说，Client对象并不知道自己处理的Shape对象是Rectangle对象还是Circle对象。于是，Client处理的具体类对于Client而言是隐藏（或者说封装）了。这是《设计模式》一书在说到“找到变化并封装之”所指的封装方式。意思是说找到变化的地方，然后将变化封装在一个抽象类“之后”（见第8章）。

我们按照这个过程对矩形绘图问题来进行设计。

试一试：找到变化之处

首先，找到什么在发生变化。在这里，变化的是形状的种类和绘图程序的种类。而共同的概念是“形状”和“绘图程序”，如图10-9所示。（请注意类名用斜体字表示，因为这两个类是抽象类。）



图10-9 什么在发生变化

现在，我希望Shape类封装形状种类的概念。形状需要知道如何绘制自己，而 Drawing 对象负责画线和圆。我通过在类中定义方法来表示这些责任。

试一试：表示变化

下一步是表示具体的变化。Shape类有矩形和圆形，绘图程序分别有一个基于DP1的对象（V1Drawing）和一个基于DP2的对象（V2Drawing），如图10-10所示。

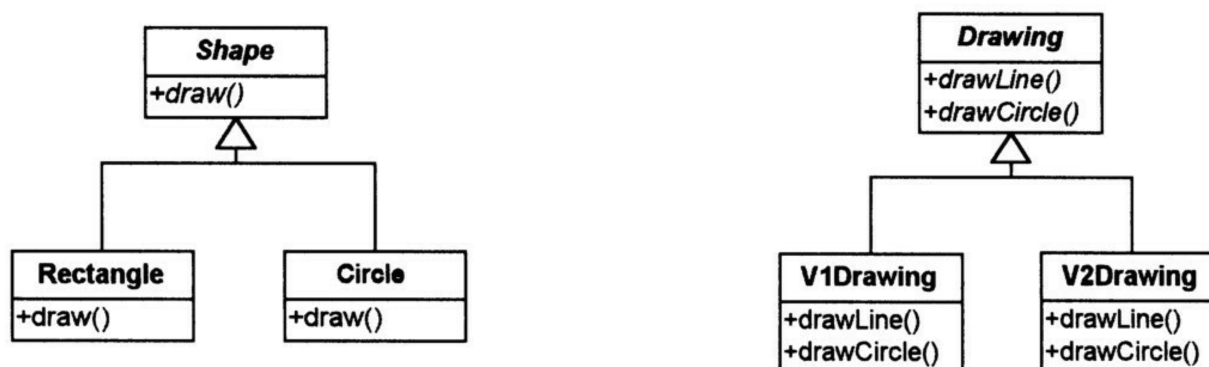


图10-10 表示变化

现在图还只是示意性的。我知道 V1Drawing 对象将使用 DP1，V2Drawing对象将使用DP2，但是还没有说明如何使用。我只是捕获了问题域的概念（形状和绘图程序），并表示了存在的变化。

将这些类联系起来：让谁使用谁？

有了这两组类之后，还需要知道它们之间如何联系。我不想再在继承树中增加一组新的类，因为我知道这样做会导致什么情况。（请查看图10-3和图 10-7，回忆一下。）相反，我想看看是否能通过让一组使用另一组，将这些类联系起来（也就是说，遵循“优先使用对象聚集，而不是类继承”的要求）。问题是，该让哪组类使用另一组类呢？

考虑如下两种可能性：Shape类使用 Drawing程序，或者 Drawing程序使用Shape类。

首先考虑后一种情形。如果绘图程序能够直接绘制形状，它们必须

对形状的一些情况有大致了解：是什么、看起来如何。但这违反了对象的一个基本原则：对象应该只对自己负责。

这样做还违反了封装。Drawing 对象要绘制形状，必须知道 Shape的具体信息（即Shape的种类）。因此对象并不是真正对自己的行为负责。

现在，考虑前一种情形。如果让 Shape对象使用 Drawing对象来绘制自己如何？Shape对象无需知道所用Drawing对象的类型，因为可以让Shape引用Drawing类。Shape对象仍然要负责控制绘图过程。

这看起来更好。这个解决方案如图10-11所示。

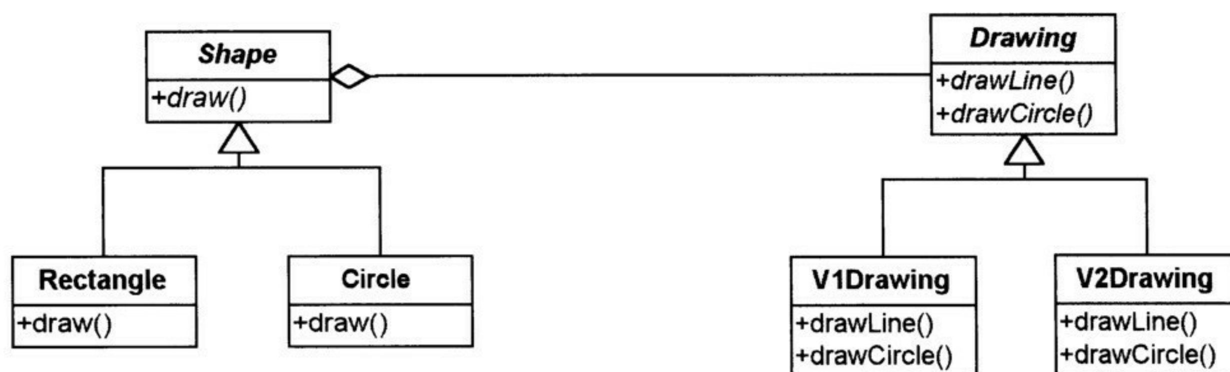


图10-11 将类联系起来

扩展设计

在这个设计中，Shape类通过Drawing类具体实现自己的行为。其中略去了V1Drawing使用DP1程序、V2Drawing使用DP2程序的细节。图10-12 中增加了这些细节，以及保护方法 drawLine 和 drawCircle（在Shape 类中），这两个方法分别调用 Drawing 的 drawLine 方法和drawCircle方法。

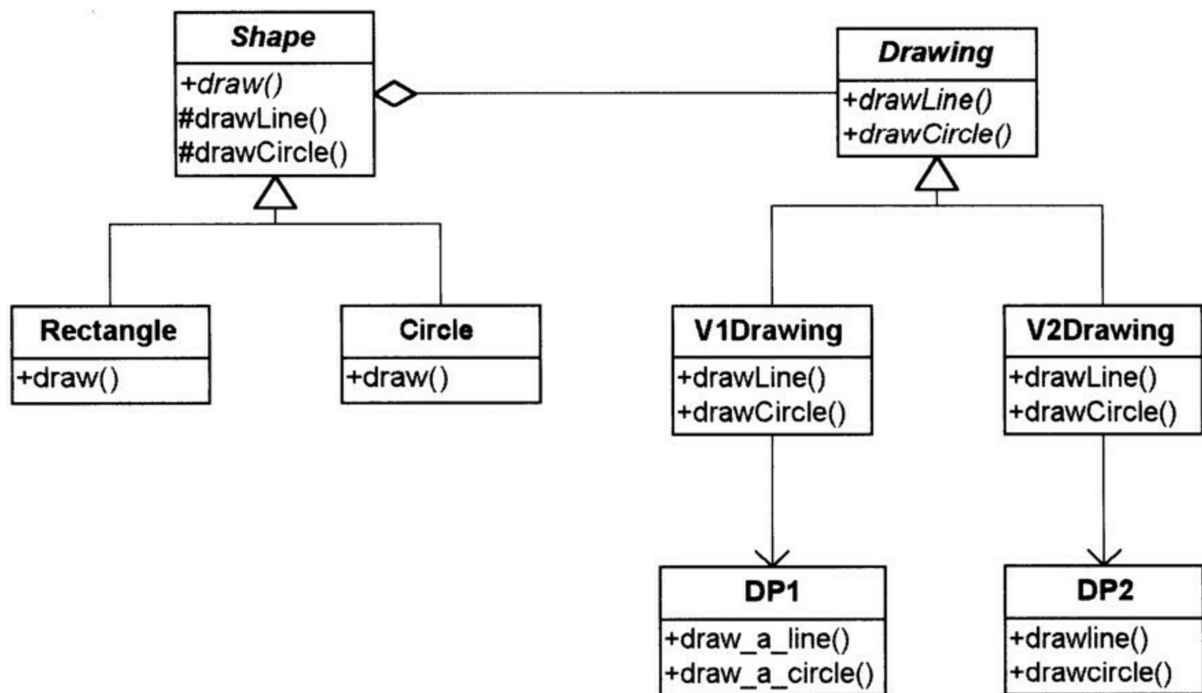


图10-12 扩展设计

模式图解

图10-13说明了Shape抽象与Drawing实现的分离。

一条规则，实现一次

有一条重要的实现策略应该遵循：规则只在一个地方实现。[35]换言之，如果做什么事情有一条规则，只实现一次。这通常会使代码中出现许多小的方法，所增加的代价很小，却消除了重复，而且经常可以预防将来可能出现的很多问题。重复的害处，不仅仅在于输入工作成倍增加，还因为将来有东西可能发生变化时，可能会忘记在所有需要地方进行修改。

虽然 draw方法或者 Rectangle类可以直接调用 Shape类的任何Drawing对象的 draw Line方法，但还是可以根据“一条规则，一个地方”策略进一步改进代码，让Shape类的drawLine方法调用自己所拥有的Drawing对象的drawLine方法。

我并不是纯粹主义者（至少大多数事情上不是），如果有什么地方我认为应该总是遵循原则的，那就是这里。下面的例子中 Shape 类有一个drawLine方法，因为它描述了用Drawing对象画线的规则。同样地，使用drawCircle方法画圆。通过遵循这种策略，也为其他可能需要绘制线和圆的派生类做好了准备。

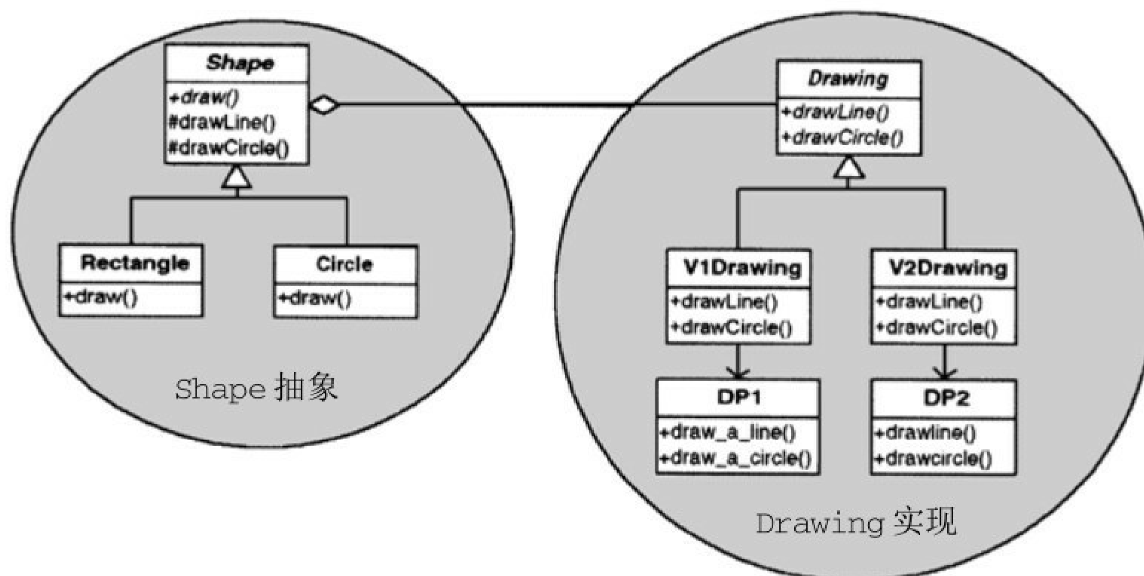


图10-13 说明“抽象与实现分离”的类图

新设计与基于继承的设计的关系

从方法的角度来看，这与基于继承的实现（如图 10-3 中所示）非常相似。最大的区别在于，方法现在被放在了不同的类中。

在本章开始曾经说到，我对Bridge模式的迷惑是由于我对术语“实现”的误解。我认为“实现”是指“如何实现一个特定抽象”。

Bridge模式使我明白，将实现看成对象之外的东西，看出由对象所使用的东西，这样就使变化隐藏在实现中，与调用程序隔离了，从而提供了极大的自由。用这种方式设计对象，还能够看到将变化包含在不同的类层次中了。图10-13中左边的类层次包含了抽象中的变化，图10-13中右边的类层次包含了实现这些抽象时包含的变化。这与前面提到的创建对象的新范型（使用共性/可变性分析）是一致的。

从对象的角度

想象一下吧，即使有很多类，但任何时候只需要同时处理3个对象，这非常直接地说明了一切（如图10-14所示）。

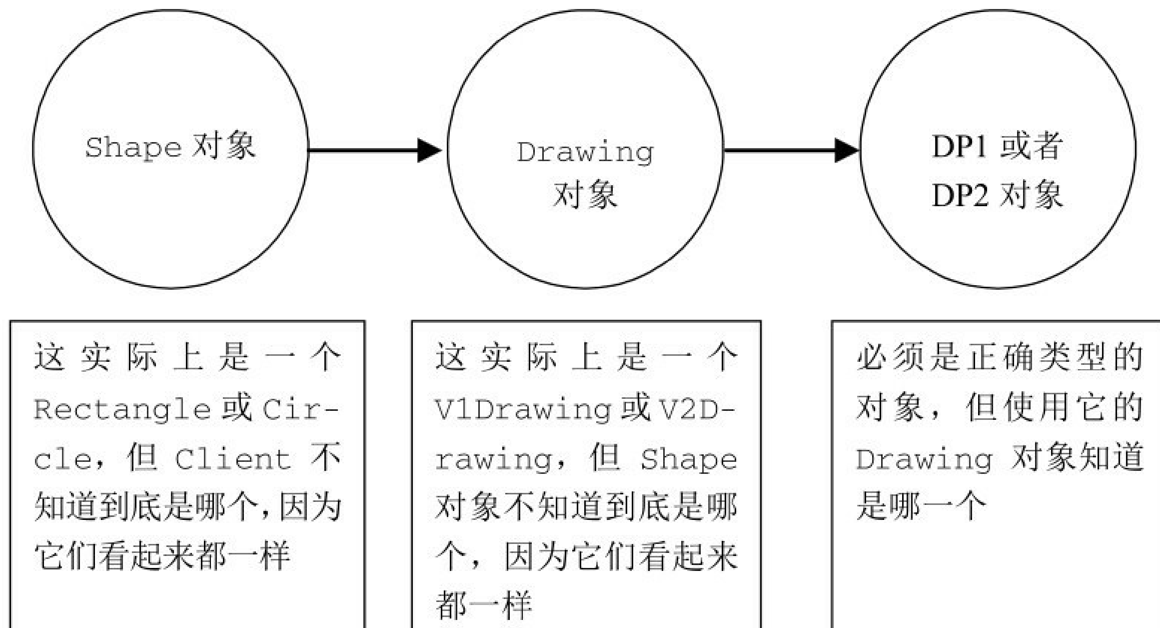


图10-14 同时只有3个对象

代码示例

例10-3给出了相对完整的Java代码。

例**10-3 Java**代码片段

```
public class Client {
    static public void main () {
        Shape myShapes[];
        Factory myFactory= new Factory();
        // 从其他地方取得矩形
        myShapes= myFactory.getShapes();
        for (int i= 0; i < myShapes.length; i++) {
```

```
            myShapes[i].draw();
        }
    }
}
abstract public class Shape {
    protected Drawing myDrawing;
    abstract public void draw();
    Shape (Drawing drawing) {
        myDrawing= drawing;
    }
    protected void drawLine (
        double x1,double y1, double x2,double y2) {
        myDrawing.drawLine(x1,y1,x2,y2);
    }
    protected void drawCircle (
        double x,double y,double r) {
        myDrawing.drawCircle(x,y,r);
    }
}
public class Rectangle extends Shape {
    private double _x1, _y1, _x2, _y2;
    public Rectangle (Drawing dp, double x1,
        double y1, double x2, double y2) {
        super( dp);
        x1= x1; _y1= y1; _x2= x2; _y2= y2;
    }
    public void draw() {
```

```
        drawLine( _x1, _y1, _x2, _y1);
        drawLine( _x2, _y1, _x2, _y2);
        drawLine( _x2, _y2, _x1, _y2);
        drawLine( _x1, _y2, _x1, _y1);
    }
    protected void drawLine(double x1, double y1,
        double x2, double y2) {
        myDrawing.drawLine( x1, y1, x2, y2);
    }
}
public class Circle extends Shape {
    private double _x, _y, _r;
    public Circle (Drawing dp,
        double x, double y, double r) {
        super(dp);
        x= x; _y= y; _r= r;
    }
    public void draw() {
        myDrawing.drawCircle( _x, _y, _r);
    }
}
abstract public class Drawing {
    abstract public void drawLine(double x1,
        double y1, double x2, double y2);
        abstract public void drawCircle(
        double x, double y, double r);
}
```

```
public class V1Drawing extends Drawing {
    public void drawLine (
        double x1,double y1,
        double x2,double y2) {
        DP1.draw_a_line(x1,y1,x2,y2);
    }
    public void drawCircle (
        double x,double y,double r) {
        DP1.draw_a_circle(x,y,r);
    }
}
public class V2Drawing extends Drawing {
    public void drawLine (
        double x1,double y1,
        double x2,double y2) {
        // DP2中的参数不同
        // 必须重排顺序
        DP2.drawLine(x1,x2,y1,y2);
    }
    public void drawCircle (
        double x, double y,double r) {
        DP2.drawCircle(x,y,r);
    }
}
```

10.6 Bridge模式回顾

Bridge模式的本质

我们已经了解了Bridge模式的工作原理，现在应该从更概念性的角度再来回顾一下。如图10-13所示，这个模式由一个抽象（及其派生）和一个实现组成。用Bridge模式进行设计时，牢牢记住这两部分将很有帮助。设计实现的接口时，应该考虑到它必须支持抽象类的不同派生类。请注意设计者设计出的接口，没有必要考虑抽象类所有可能的派生类（这可能导致另一种分析瘫痪），而只需要支持那些实际要构造的派生类即可。我们不断看到仅仅是这种对灵活性的考虑，就经常能够极大地改善设计。

注意：在C++中，必须用一个定义公用接口的抽象类来实现Bridge模式的实现部分。在C#和Java中，既可以使用抽象类，也可以使用接口，具体选择哪一种取决于实现是否具有抽象类都能利用的特性。[36]

10.7 实践注记：使用Bridge模式

经典的 Bridge 模式例子

打印驱动程序可能是 Bridge 模式最典型的例子，也是最适合应用Bridge模式的场合。但是在我看来，Bridge模式的真正威力在于它能够帮助我看到什么时候应该从问题域中提取实现。也就是说，有时候有一个实体X使用系统S，一个实体Y使用系统T。我可能认为X总是与S相伴，而Y总是与T 相伴，因此就将它们联系（耦合）起来。Bridge模式提醒我，可以抽象出S和T（X和Y的实现）之间的区别，从而使X和Y都可以使用S和T，这样更好。也就是说，Bridge模式最有用的地方，是在解耦抽象与实现之前考虑Bridge模式是否适用。

Birdge 模式经常与Adapter模式结合

请注意，图10-12和图10-13中所示的解决方案结合使用了Bridge模式和Adapter模式。这样做是因为必须使用给定的绘图程序，绘图程序

有已经存在的接口必须遵循。因此需要使用Adapter模式先进行适配，然后才能用同样的方式处理它们。

尽管Adapter模式与Bridge模式相结合的情况非常常见，但Adapter模式并不是Bridge模式的一部分。

复合设计模式

当两个或两个以上的模式紧密结合（就像上面的Bridge模式和Adapter模式）时，就形成了所谓“复合设计模式”[37]。现在可以讨论“模式的模式”了。

实例化 Bridge 模式的对象

还需要注意的是：表示抽象的对象（比如Shape对象）是在实例化时实现。这不是Bridge模式本身要求的，却很普遍。

Bridge模式：关键特征

意图

将一组实现与另一组使用它们的对象分离。

问题

一个抽象类的派生类必须使用多个实现，但不能出现类数量爆炸性增长。

解决方案

为所有实现定义一个接口，供抽象类的所有派生类使用。

参与者与协作者

Abstraction为要实现的对象定义接口，Implementor为具体的实现类定义接口。Abstraction的派生类使用Implementor的派生类，却无需知道自己具体使用哪一个ConcreteImplementor。

效果

实现与使用实现的对象解耦，提供了可扩展性，客户对象无需操心

实现问题。

实现

将实现封装在一个抽象类中。

在要实现的抽象的基类中包含一个实现的句柄。注意：在Java中，你可以在实现中使用接口来代替抽象类。

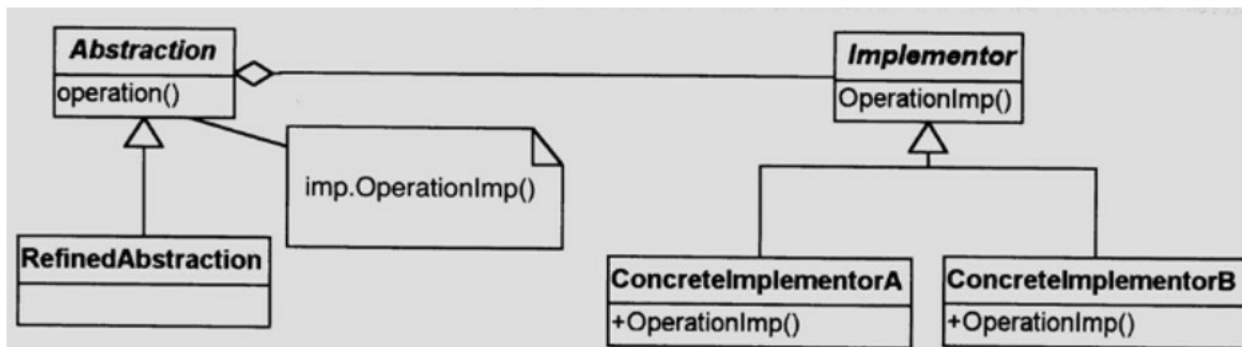


图10-15 Bridge模式的一般结构图

既然已经理解了Bridge模式，现在我们来回顾《设计模式》一书中关于这个模式描述的“实现”一节。其中讨论了关于“抽象如何创建和/或使用实现”的各种问题。

在Birdge模式中，C#和 Java 相对于 C++的一个优点

在使用Bridge模式时，几个抽象对象有时候可能共享实现对象。

在C#和Java中，这不会有什么问题；当所有的抽象对象销毁之后，垃圾回收器会发现不再需要实现对象，并清理它们。

在 C++中，我必须自己管理实现对象。办法有很多：可能是维护一个引用计数器，甚或使用Singleton模式。但是，如果能够不考虑这个问题当然更好。这说明了自动垃圾回收的另一个优点。

虽然使用 Bridge 模式开发的解决方案大大优于原方案，但是并不完美。衡量设计质量的一种方法是：看它是否能很好地应对变化。采用了Bridge模式，处理新的实现非常容易。程序员只需要定义一个新的具体实现类，并实现它即可，不需要修改任何其他东西。

Bridge 模式解决方案很好，但并不总是完美的

但是，如果抽象有一个新的具体情况，事情就没那么轻松了。对于某些新的Shape，可以用设计中已有实现来实现。但是，也可能有一种新的Shape，需要新的绘图函数，比如，可能需要实现椭圆。现在的Drawing类没有用来画椭圆的方法，这种情况下，必须对实现进行修改。但是，即使真地出现了这种情况，至少修改过程是明确定义的：修改 Drawing 类的接口，再相应地修改所有 Drawing 派生类。这种修改过程将变化的影响面控制在局部，并且降低了出现副作用的风险。

我们得到的是一个优秀的解决方案，虽然并不完美。而且Bridge模式给了我处理问题的方法，可以用于思考更通用的实现。设计模式能够帮助我对解决方案进行更抽象和更通用的思考。我可以自己决定是否用更通用的解决方案来实现，设计模式对此并不强制要求。

结论：模式并不总能提供十全十美的解决方案。但是因为模式是众多设计人员多年的集体经验结晶，所以它们通常优于你我自己在有限的时间所能提出的解决方案。

遵循“一条规则，实现一次”策略有助于重构。

在真实的世界中，不会总是一开始就有很多实现。有时虽然明知可能存在新的实现，但它们总是在意想不到的时候出现。一种办法，就是总是使用抽象，为多个实现做好准备。这样就能获得非常通用的应用程序。

但我并不推荐这种办法，这会使类的数量不必要地增加。在真的出现多个实现时（经常如此），用这种办法编写代码是很重要的，因为这样修改这些代码以采用Bridge模式并不困难。修改代码改进结构但不增加新功能，就是所谓重构（refactoring）。正如 Martin Fowler所定义的：“重构是一种修改软件系统的过程，它在不改变代码的外在行为的情况下，改进它的内部结构。”[38]

在设计代码时，我总是遵循“一条规则，实现一次”的要求，从而密切留意重构的可能性。drawLine 方法是一个好例子。虽然代码实际实现

的位置发生了改变，但是移动起来非常容易。

重构

重构在面向对象设计中很常用。但是严格地说来，它并不仅仅局限于面向对象……在不增加功能的情况下修改代码以改进结构，都可以称为重构。

观察Bridge模式的一种有用方式

在我们对Bridge模式的推演过程中，出现了两个变化（形状和绘图程序），它们都被封装在各自的抽象类中。也就是说，形状的变化封装在Shape类中，绘图程序的变化封装在Drawing类中。

回来再看看这两个多态结构，我问自己："这两个抽象类表示的是什么呢？"对于形状而言，显然Shape抽象类表示不同种类的形状；Drawing抽象类表示实现Shape类的方式；模式所体现的是不同抽象之间的关系。因此，在前面提到的出现 Drawing 类新需求的情况下（比如需要实现椭圆），类之间的关系仍然清晰，如何实现一目了然。

10.8 小结

本章内容

在考察Bridge模式的同时，我们研究了一个问题，在它的问题域中有两种变化——形状和绘图程序。问题域中它们各自都会发生变化。在尝试根据所有存在的特殊情况来实现解决方案时，我遇到了难题。原来的解决方案很自然地过度使用了继承，得到的设计冗余、紧耦合且低内聚、难以维护。

在学习Bridge模式的过程中，我们知道应该遵循如下应对变化的基本策略：

找到变化并封装之；

优先使用对象聚集而不是类继承。

“找到变化”在了解问题域的过程中永远是一个重要的步骤。在绘图程序的例子中，我让一组变化量使用另一组变化量，这说明可能适用于Bridge模式。

一般而言，应该将设计模式与问题域的特性和行为进行匹配，以确定应该使用哪个模式。理解了你工具库中模式的“其然”和“所以然”之后，选择适用的模式时将更加高效。你可以在决定如何实现模式之前先考虑选择哪个模式。

考虑与使用

我在前面这句话中使用了“考虑”一词代替“使用”，是有意为之。实际上，“使用”模式时应该“考虑”模式所蕴涵的问题和与模式相关的各种知识。糟糕的是，但人们听到“使用”模式之类的话时，他们很容易理解为“使用模式的实现”。采用“考虑”这个词，有助于人们认识到应该将模式看作是一种指导，一个由许多考虑事项组成的列表。

使用[39]Bridge模式之后，设计和实现都将更加坚实，能更好地应对未来的变化。

Bridge 模式中使用的面向对象原则的总结

虽然本章中主要讨论的是Bridge模式，但Bridge模式中使用的几个面向对象原则值得重点指出。

概　念	讨　论
对象对自己负责	有不同种类的 Shape，但它们都可以自己绘制自己（通过 draw 方法）。Drawing 类负责对象的绘图元素
抽象类	我用抽象类来表示概念。实际上问题域中有矩形和圆形。概念“形状”仅仅存在于我们的脑子里，它是将两个概念合而为一的一种手段；所以，我用 Shape 这个抽象类表示形状的概念。Shape 类永远不会被实例化，因为它在问题域中并不存在（只存在 Rectangle 类和 Circle 类）。绘图程序也是一样
通过抽象类进行封装	在这个问题中，有两个通过使用抽象类进行封装的例子 ❑ 使用 Bridge 模式的客户只能看见 Shape 类的派生。但是，客户将不知道自己拥有哪一种 Shape 对象（对于客户而言，它只是一个 Shape 对象），这样，这一信息就被封装了。这样做的优点是，如果未来需要有一种新的 Shape，将不会影响该客户对象 ❑ Drawing 类对 Shape 类隐藏了不同的 Drawing 派生类。在实践中，抽象可以知道自己使用的实现，因为抽象可以实例化实现。参见《设计模式：可复用面向对象软件的基础》，其中解释了为什么应该如此。但是，即使在这种情况下，有关实现的知识也应限制在抽象的构造函数中，以便容易修改
一条规则，实现一次	抽象类中经常有些方法实际使用实现对象，抽象类的子类会调用这些方法。这样在需要修改时，修改起来比较容易，而且在实现整个模式之前能够有一个好的起点
可测试性	想象一下，为原解决方案和后来改进后的解决方案的形状和绘图程序编写测试程序。例如，假设有 N 种形状和 M 种实现。第一种解决方案需要 N*M 个测试，而第二种解决方案需要 M+N 种测试。首先测试 M 种实现，然后用任意选择的实现测试 N 种形状（因为所有形状使用所有实现的方式都是一样的）

复习题

简答题

1.定义解耦和抽象。

2.在Bridge模式的上下文中实现是怎样定义的？

3.顺序图的基本要素是什么？

4.Alexander 怎样看待如何使用模式？他提倡先从解决方案入手还是从要解决的问题入手？

5.共性分析要寻找的是什么？可变性分析呢？

6.Bridge模式要解决的基本问题是什么？

7.给出“一条规则，实现一次”策略的定义。

8.采用Bridge模式的效果是什么？

阐述题

1.《设计模式》一书说，Bridge模式的意图是“将抽象与其实现解耦，使它们都可以独立地变化”。

这是什么意思？

给出一个例子。

2.为什么紧耦合会导致类数量的爆炸性增长？

观点与应用题

1.“用对象的职责而不是其行为来思考问题。”这对你就面向对象系统中继承的看法有什么影响？

2.你认为为什么《设计模式》一书将这个模式称为“Bridge”？就其功能而言这个名字合适吗？为什么？

第11章 Abstract Factory模式

11.1 概览

本章内容

我将以Abstract Factory（抽象工厂）模式来继续我们的模式学习之旅，这是一个用于创建一组对象的模式。

在本章中，我们将：

提供一个例子，帮助你推导出Abstract Factory模式；

给出Abstract Factory模式的关键特征；

将Abstract Factory模式与我们的CAD/CAM问题联系起来。

11.2 Abstract Factory模式简介

意图：协调对象的实例化

《设计模式》一书中对Abstract Factory模式的意图是这样叙述的：“为创建一组相关或相互依赖的对象提供一个接口，而且无需指定它们的具体类”。[40]

有时候，几个对象需要以一种协调的方式实例化。例如，在处理用户界面时，系统可能需要在一个操作系统上用一组对象，在另一个操作系统上用另一组对象。Abstract Factory模式能够确保系统总是根据情况获得正确的对象。

11.3 学习Abstract Factory模式：示例

一个引出问题的例子：根据机器能力选择设备驱动程序

假设派给我这样一项任务：设计一个计算机系统，显示并打印取自数据库的几何形状。用来显示和打印形状的分辨率类型取决于当前运行系统的计算机：CPU的速度和可用内存。系统必须留意自己对计算机的要求。

这里的难点在于，系统必须控制使用哪些驱动程序：低配置机器使用低分辨率驱动程序，高配置机器使用高分辨率驱动程序，如表11-1所示。

表11-1 不同机器的不同驱动程序

驱动功能	在低配置机器上，使用……	在高配置机器上，使用……
显示	LRDD 低分辨率显示驱动程序	HRDD 高分辨率显示驱动程序
打印	LRPD 低分辨率打印驱动程序	HRPD 高分辨率打印驱动程序

根据共同的概念定义不同的组

这个例子中，两组驱动程序是互斥的，但通常实际情况并不是这样。有时候不同组的驱动程序会包含来自同一个类的对象。比如，一台中级配置的机器可能使用低分辨率显示驱动程序（LRDD）和高分辨率打印驱动程序（HRPD）。

使用哪个组取决于问题域：对于给定的情形，需要使用哪组对象？在本例中，共同的概念主要是对象对系统提出的要求。

低分辨率组——LRDD和LRPD，这些驱动程序对系统提出的要求较低。

高分辨率组——HRDD 和 HRPD，这些驱动程序对系统提出的要求较高。

方案1：使用 switch语句选择驱动程序

我的第一次尝试可能是用switch语句控制驱动程序的选择，如例11-1所示。

例**11-1 Java**代码片段：**switch**语句控制所用的驱动程序

```
// Java代码片段
class ApControl {
   ...
   public void doDraw() {
      ...
      switch (RESOLUTION) {
         case LOW:
            // 使用 lrdd
         case HIGH:
            // 使用 hrdd
      }
   }
   public void doPrint() {
      ...
      switch (RESOLUTION) {
      case LOW:
         // 使用 lrpd
      case HIGH:
         // 使用 hrpd
      }
   }
}
```

……但是耦合度和内聚性上都有问题

虽然这样能够达到目的，但是有问题。确定该使用哪个驱动程序的

规则与驱动程序的实际使用混杂在一起，因此耦合度和内聚性上都存在问题。

紧耦合——如果要修改分辨率的规则（比如需要添加一个MIDDLE值），就必须在两处修改代码，而这两处的其他方面毫无关系。

低内聚——doDraw方法和doPrint方法的任务毫不相关：它们都必须创建形状，并且操心应该使用哪个驱动程序。

紧耦合和低内聚也许眼下并不是问题，但是，它们通常会增加维护成本。此外，在实践中，不仅仅是这里指出的两处，可能更多的地方都会受影响。

switch语句可能说明需要抽象

switch语句本身常常说明：（1）需要多态行为；（2）存在职责错放。应该考虑用一种更通用的解决方案，比如抽象代替switch语句，或者将职责赋予其他对象。

方案2：使用继承

另一种方案是使用继承。可以用两个不同的 ApControl 类：一个使用低分辨率驱动程序，另一个使用高分辨率驱动程序。它们都将从同一个抽象类派生，因此可以在抽象类中维护公共代码，如图11-1所示。

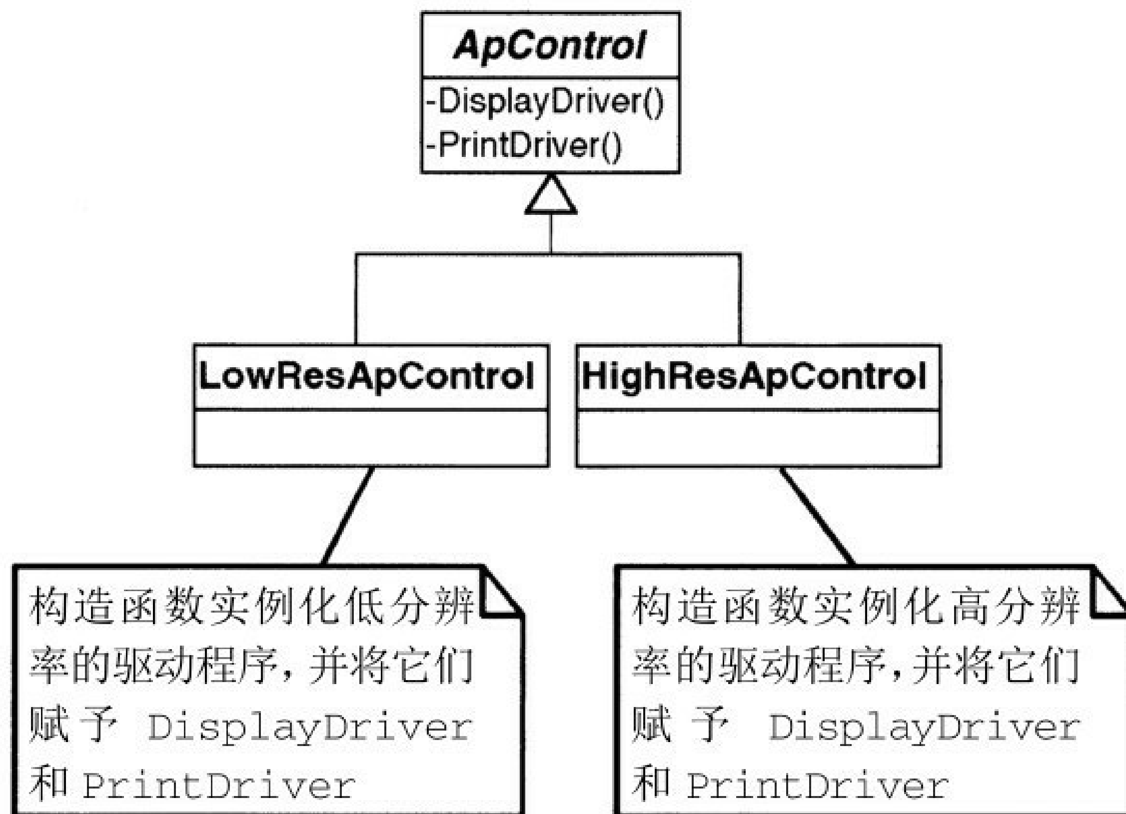


图11-1 方案2——用继承处理变化

……但这种方案也有问题

虽然在这个简单的例子里使用继承也可以完成任务，但继承的缺点非常之多，我甚至宁愿使用switch语句。略举一二吧。

组合爆炸——对于每个组和未来可能使用的每个新组，都必须创建一个新的具体类（即ApControl的一个新版本）。例如，如果需要中级配置的应用程序（使用 LRPD 和 HRDD），就需要增加一个新类处理这一情况。如果需要一个应用程序处理HRPD和LRDD，又需要另一个类。这样，图11-1中ApControl将有许多派生类。

含义不清——所生成的类对于说明意图毫无帮助。我已经将每个类根据特定情况进行了特化。如果希望自己的代码未来维护起来比较容易，就需要尽可能清晰地说明其意图。这样就不必再耗费大量时间重新了解某段代码的用途。

需要使用聚集——这个方案居然违反了“优先使用对象聚集而不是类继承”的基本规则。没有遵循这一规则说明在发生其他变化时，这些类会在类层次中进一步降级。

方案3：用抽象来代替switch语句

根据我的经验，switch语句经常意味着可能应该使用抽象。在本例子中，LRDD和HRDD都是显示驱动程序，LRPD和HRPD都是打印驱动程序。所以抽象概念应该就是显示驱动程序和打印驱动程序。图 11-2概念性地表示了这一点。之所以说是“概念性地”，是因为LRDD和HRDD并不是从同一个抽象类派生出来的。现在我用不着操心“LRDD和HRDD派生自不同的抽象类”，因为我知道可以用Adapter模式对这些驱动程序进行适配，使它们看起来属于一个抽象类。

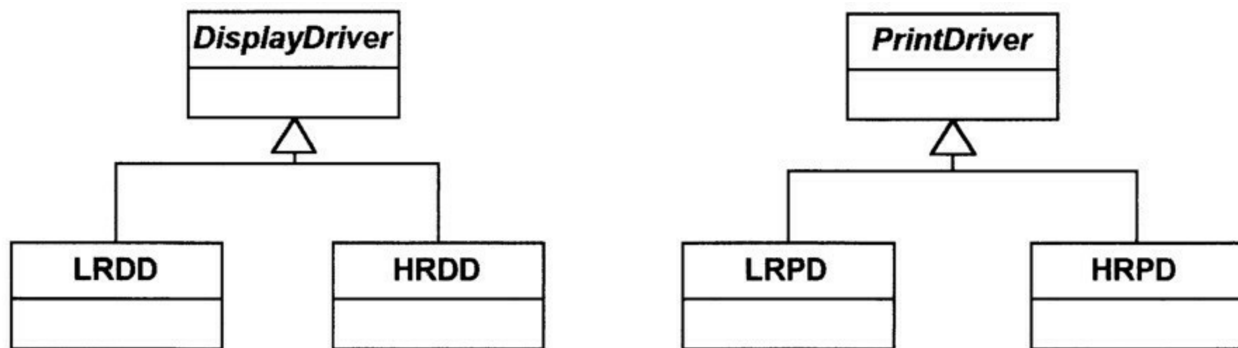


图11-2 驱动程序及其抽象

代码更易理解

这样定义对象，ApControl 对象不用 switch 语句就可以使用DisplayDriver对象和PrintDriver对象了。ApControl类的可理解性大大提高了，因为它自己用不着再考虑驱动程序的具体类型。也就是说，ApControl对象可以不考虑驱动程序的分辨率，使用DisplayDriver对象和PrintDriver对象。ApControl使用驱动程序很可能又可以实现Briolge模式。我们将使用Abstract factory模式（现在还没有定义）为此做好准备。

参见图11-3和例11-2中的代码。

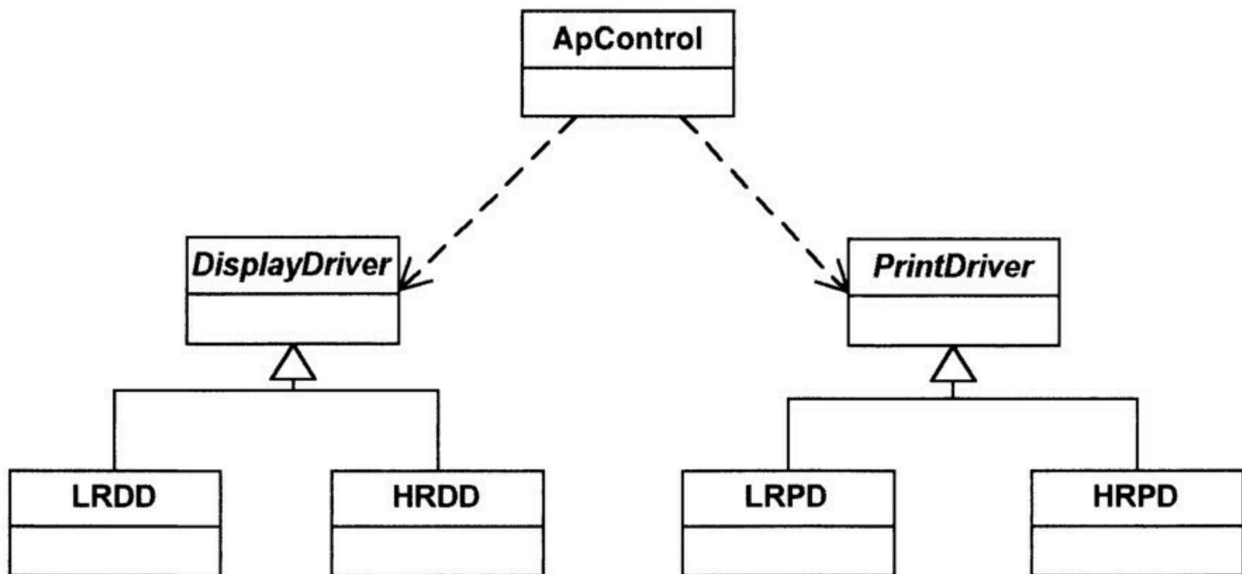


图11-3 理想情况下ApControl使用驱动程序的情形

例**11-2 Java**代码片段：使用多态解决这一问题

```
// Java代码片段
class ApControl {
   ...
   public void doDraw() {
      ...
      myDisplayDriver.draw();
   }
   public void doPrint() {
      ...
      myPrintDriver.print();
   }
}
```

工厂对象

还有一个问题：怎样创建合适的对象呢？

可以让 ApControl 类负责，但这样未来会有维护问题。如果要处理一组新的对象，就必须修改 ApControl 类。相反，如果用一个“工厂”对象负责实例化需要的对象，即使出现新的对象组也没有问题了。

在本例中，我将用一个工厂对象（ResFactory类型，也称分辨率工厂）来控制驱动程序组的创建。ApControl对象将使用另外一个对象——工厂对象获得适合当前所用计算机的显示驱动程序和打印驱动程序。具体交互过程如图11-4所示。

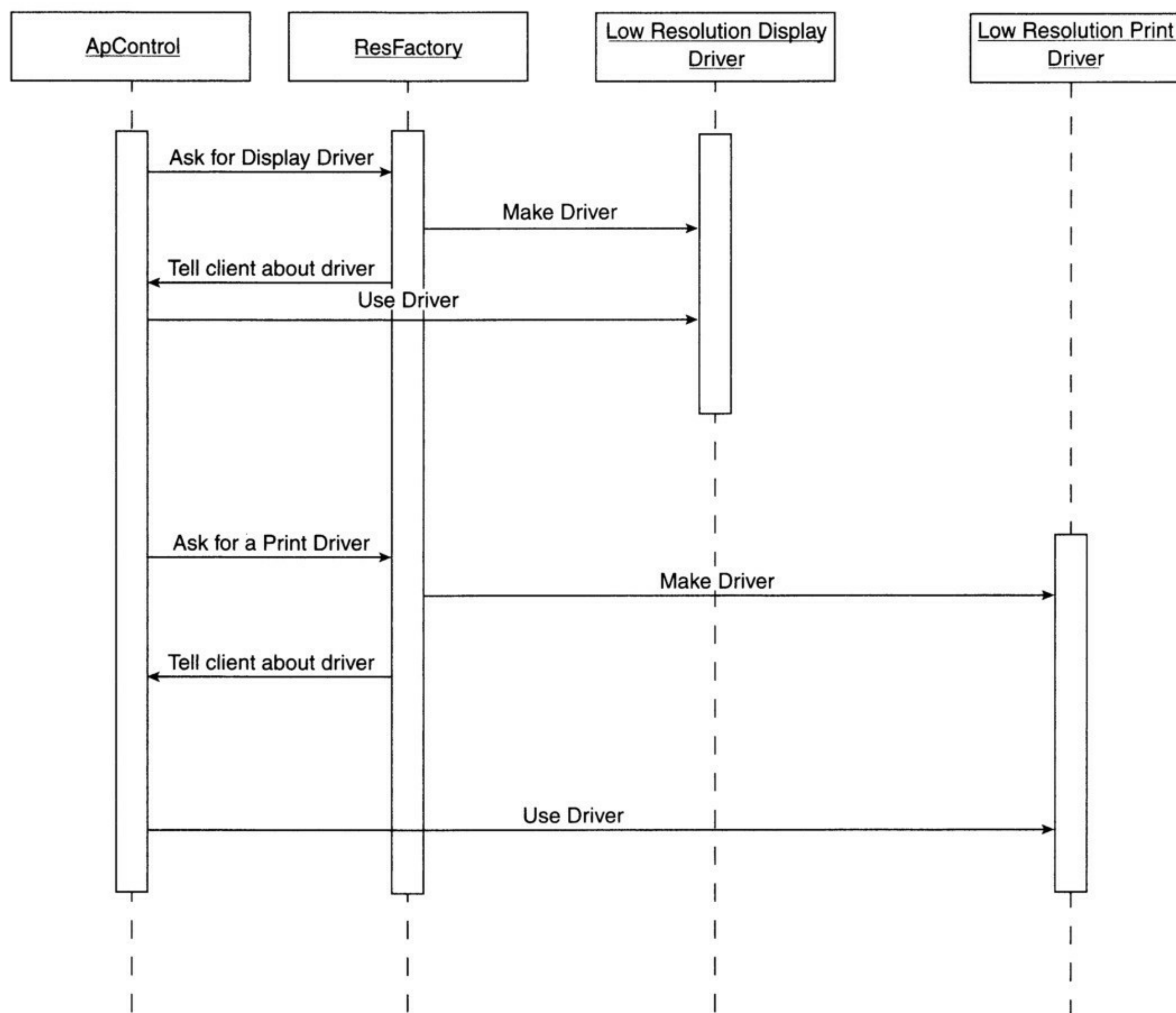


图11-4 ApControl对象从一个工厂对象获取驱动程序

工厂是有一定职责的……而且是内聚的。

从 ApControl 的角度来看，事情现在简单多了。ApControl 让 ResFactory 来负责跟踪应该使用哪些驱动程序。虽然我们仍然需要为 ResFactory 编写选择代码，但是问题已经根据职责分解了。ApControl的职责是了解如何使用合适的对象。ResFactory的职责是决定哪些对象合适。可以使用不同的工厂对象，也可以只使用一个（这时可能需要使用switch语句）。无论如何，现在的情况都优于前面的方案。

这同时还加强了内聚性：ResFactory所做的就是创建合适的驱动程

序；ApControl只负责使用这些驱动程序。

……将变化封装在一个类中

避免在ResFactory类中使用switch语句的办法有几种。不使用switch语句可以在未来进行修改时，不影响原有的工厂对象。可以通过定义一个表示“工厂”概念的抽象类将变化封装在一个类中。对于ResFactory而言，有两种不同的行为（方法）。

给我应该使用的显示驱动程序。

给我应该使用的打印驱动程序。

ResFactory对象可以从两个具体类中的一个实例化，而具体类都从一个定义了公共方法的抽象类派生，如图11-5所示。

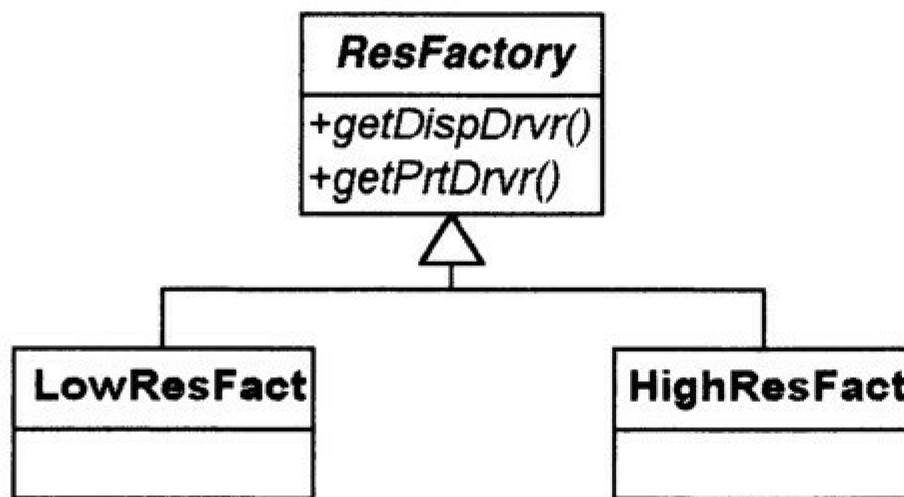


图11-5 ResFactory类封装了变化

将分析和设计联系起来的方法

下面是Abstract Factory模式的3个关键的概念步骤。

策略	设计中的体现
找到变化并封装之	使用哪个驱动程序对象的选择是变化的，所以，将它封装在ResFactory类中。
优先使用对象聚集，而不是类继承	将变化放在一个独立的对象——ResFactory对象中，ApControl对象使用ResFactory对象，而不是拥有两种不同的ApControl对象。
针对接口而不是实现设计	ApControl知道怎样请求ResFactory实例化驱动程序对象，但它不知道（或者说无需操心）ResFactory对象如何实际实例化。

11.4 学习Abstract Factory模式：实现该模式

实现设计

例11-3展示了这个设计中的Abstract Factory模式是如何实现的。

例**11-3 Java**代码片段：实现**ResFactory**

```
abstract class ResFactory {
   abstract public DisplayDriver getDispDrvr();
   abstract public PrintDriver getPrtDrvr();
}
class LowResFact extends ResFactory {
   public DisplayDriver getDispDrvr() {
      return new LRDD();
   }
```

```
    public PrintDriver getPrtDrvr() {
      return new LRPD();
    }
}
class HighResFact extends ResFactory {
    public DisplayDriver getDispDrvr() {
      return new HRDD();
    }
    public PrintDriver getPrtDrvr() {
      return new HRPD();
    }
}
```

集成：Abstract Fac-tory模式

我们来完成这个解决方案，让 ApControl 对象与合适的工厂对象（LowResFact对象或HighResFact对象）通信，如图11-6所示。请注意，ResFactory是抽象的，这种对 ResFactory实现细节的隐藏，正是这个模式的运作原理。因此，这个模式被称为Abstract Factory（抽象工厂）。

工作原理

ApControl对象将得到一个 LowResFact对象或一个 HighResFact对象。需要时 ApControl 对象向这个工厂请求对象合适的驱动程序。工厂对象实例化自己了解的某个驱动程序对象（低分辨率或高分辨率）。ApControl对象不需要操心返回的驱动程序是低分辨率还是高分辨率，因为ApControl对象使用它们的方式相同。

LRDD/LHDD 对和LRPD/HRPD 对不需要从同一类中派生

这里还忽略了一个问题：LRDD 和 HRDD 可能不是派生自同一个抽象类（LRPD和HRPD就是如此）。我们已经学习了Adapter模式，所以这不是什么大问题。我们可以使用图 11-6 中的结构，但需要对驱动

程序进行适配，如图11-7所示。

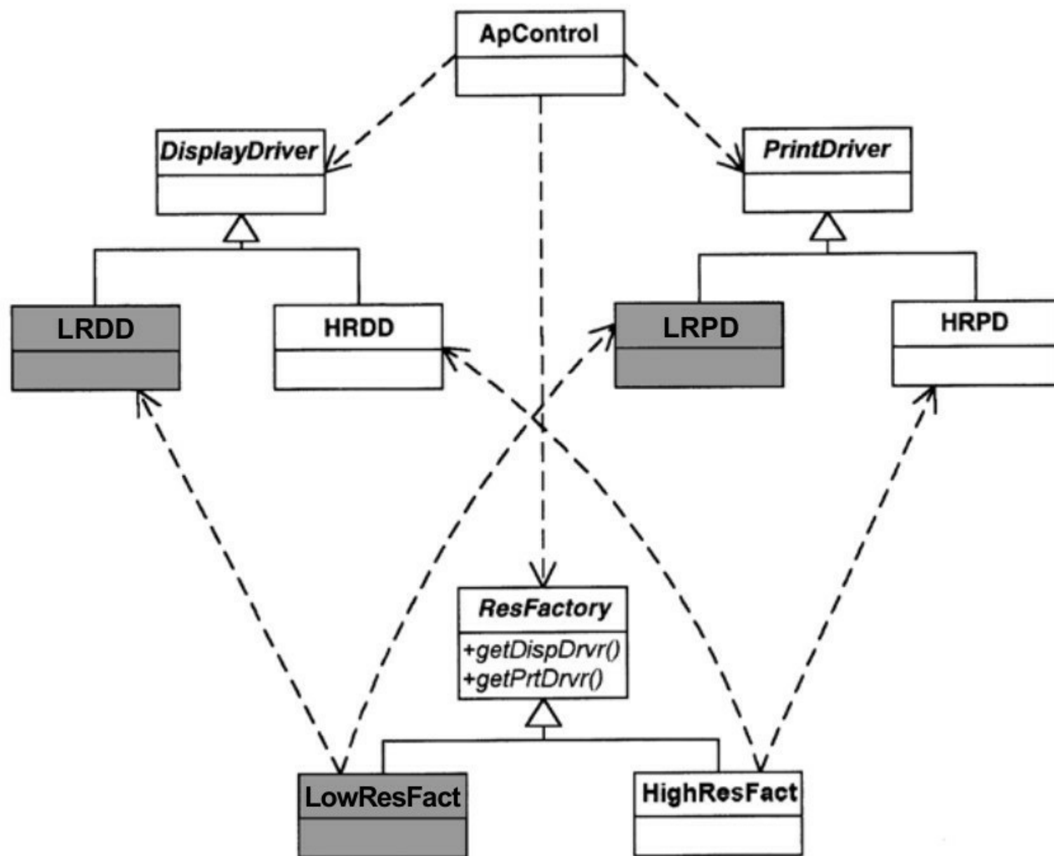


图11-6 使用Abstract Factory 模式的过渡解决方案

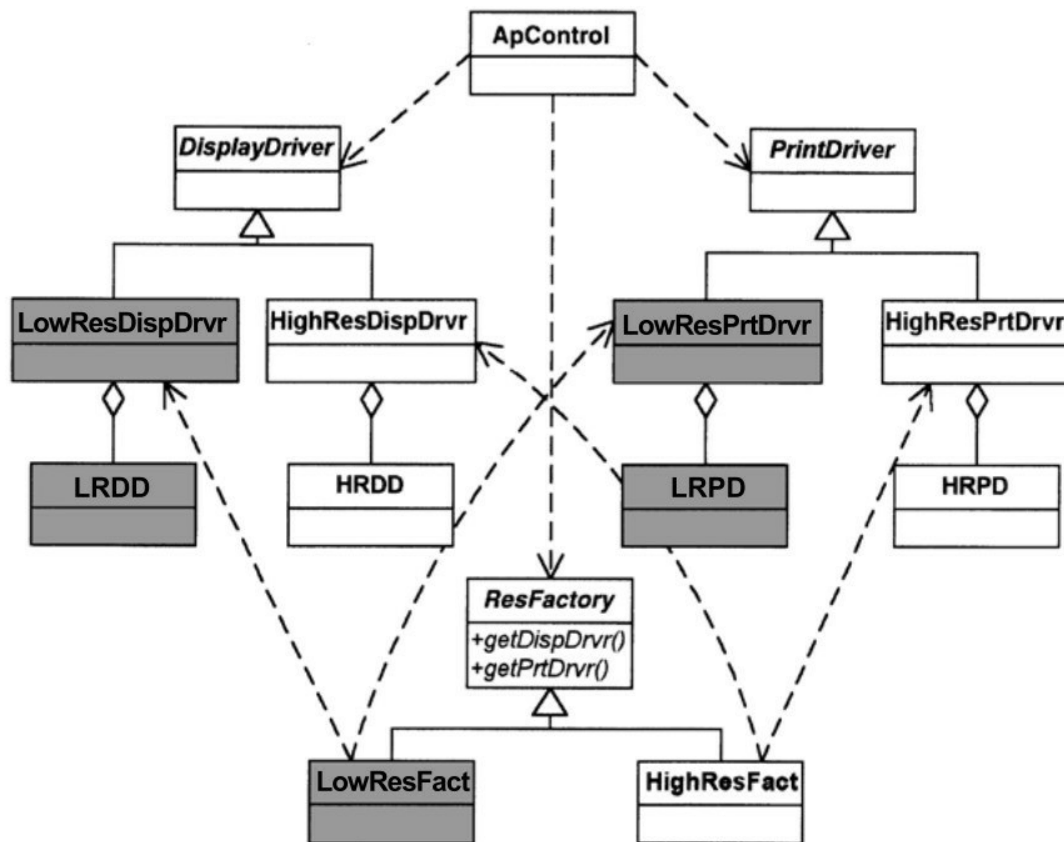


图11-7 用Abstract Factory 模式和Adapter模式解决这个问题

工作原理

这个设计的实现基本上与前面那个相同。唯一区别在于，现在工厂对象实例化的对象来自另外两个适配原驱动程序的类。这是一个重要的建模方法。用这种方法将Adapter模式和Abstract Factory模式结合在一起，我可以将概念上相似的对象当作同种对象处理，即使它们不是。这使Abstract Factory 模式可以用于更多情况。

Abstract Factory模式中对象的角色

在这个模式中：

客户对象只知道向谁请求所需的对象和如何使用这些对象；

Abstract Factory类通过为每个不同类型的对象定义一个方法，来指定实例化哪个对象，一般而言，对于每一种必须实例化的对象，Abstract Factory对象都有一个相应的方法；

具体的工厂对象指定哪些对象要实例化。

实践中的Abstract Factory模式

刚才给出的例子通过为每一种可能情况设一个具体类，实现了Abstract Factory模式。这里有两种情况：低分辨率和高分辨率。实践中，情况的数量可能非常大，而且每增加一个新的变量（即新的组成员）都会组合级数地增长。

例如，在一个电子商务系统中每个客户都可能有相关的对象“组”。也就是说，每组对象代表特定客户做出的选择。这种情况下，每个“组”都有一个具体类是完全行不通的。更好的解决方案是让一个类创建所有组。每个组成员可能仍然有一个相关方法，但是创建某个对象的决策应该用一条switch语句处理。问题于是就变成了：“你怎么知道每个客户的选择呢？”也就是说，switch变量的值应该是多少呢？

当然，这个问题始终存在。Abstract Factory模式并不能消除这一问题。相反，它能够告诉我们怎样做。这个模式告诉我们，将选择放在一个对象中，让这个对象负责创建要使用的对象，并使它与使用这些对象的对象分离开来。该模式还告诉我们，将对象的使用与对象的构造分离开来。这些我们将在第18章中进一步讨论。

所以，这个模式的问题如下。

都有什么情况？

怎样管理这一信息？

构造逻辑放在哪里？

刚才说过，这些问题始终存在，以前只是被其他问题所遮住了而已。考虑一下，原来的方案中和使用了Abstract Factory模式的方案中对这些问题的处理有什么不同。

都有什么情况？

原方案：ApControl知道。

Abstract Factory模式的方案：用一个配置文件说明该创建哪个具体

工厂对象。

怎样管理这一信息？

原方案：由ApControl管理。

Abstract Factory 模式的方案：每个具体对象知道应该创建哪些对象。

构造逻辑放在哪里？

原方案：由ApControl管理。

Abstract Factory模式的方案：在工厂对象中。

在电子商务解决方案中，管理信息（即有哪个客户，或者有哪组客户）要复杂一些。我们可以给工厂对象传入客户 ID，由它解决。可能的解决方案有很多种：

工厂对象可以查看配置文件或者数据库以获得这一信息；

在Web程序中，信息可能保存在cookie中。

Abstract Factory模式并没有告诉我们如何解决这些问题，它只是告诉我们将这些问题从使用对象组的对象中移出。这既解除了实例化逻辑和使用逻辑的耦合，又加强了内聚性。加强内聚性能够使主体代码可读性更强，封装了选择对象的规则，而且鼓励以更抽象的方式使用具体对象。

这不是又走回使用switch语句的老路上了吗？

经常有人问我这样的问题："你这不是又走回使用switch语句的老路上去了吗？"确实如此。但是 switch 语句本身并没有什么问题。只有在switch语句互相耦合，比如许多switch语句使用一个变量作为开关变量时，才有问题。这种连接点会变成一种依赖性，带来复杂和bug。

任何情况下，工厂代码都能很好地隔离。无论使用什么方法实现，都不会影响要简化的代码。这还意味着，我的编程方法不用那么小心谨慎，因为代码短得多，而且与其他逻辑是解耦的。

为什么称之为 Abs-tract Factory？

为什么《设计模式》一书的作者称这个模式为Abstract Factory呢？初看起来，很容易认为这是因为工厂是用每种情况有一个派生类的抽象类实现的。但事实并非如此。这个模式之所以称为Abstract Factory，是因为它要创建的东西本身是由抽象定义的（在上例中是 DisplayDriver和PrintDriver）。工厂各种变化的实现如何选择，模式并没有具体规定。

11.5 实践注记：Abstract Factory模式

如何得到正确的工厂对象

决定需要哪个工厂对象实际上与确定使用哪一组对象是相同的。例如，在前面的驱动程序问题中，有一组低分辨率驱动程序和一组高分辨率驱动程序。我怎样才能知道自己需要哪一组呢？在类似这样的例子中，很可能通过一个配置文件获知这一信息。然后可以编写几行代码，根据配置信息将合适的工厂对象实例化。

我还可以使用 Abstract Factory 模式，不同应用程序都使用同一子系统。在这种情况下，工厂对象将传给子系统，告诉子系统将要使用哪些对象。此时，通常主系统知道子系统需要哪一组对象。在调用子系统之前，将实例化正确的工厂对象。

Abstract Factory模式的工作原理及优点

图 11-8 中的 Client 对象使用派生自两个不同服务类（Abstract-ProductA和 AbstractProductB）的对象。这个设计非常简化，隐藏了实现细节，系统可维护性也更好。

Client对象不知道自己拥有的是服务对象的哪个特定具体实现，因为创建服务对象是工厂对象的职责。

Client对象甚至不知道自己使用的是哪个特定工厂，因为它只知道自己有一个Abstrac-Factory对象。它可能有一个Concrete-Factory1对象或

一个ConcreteFactory2对象，但它不知道到底是哪一个。

Abstract Factory模式：关键特征

意图

需要为特定的客户（或情况）提供对象组。

问题

需要实例化一组相关的对象。

解决方案

协调对象组的创建。提供一种方式，将如何执行对象实例化的规则从使用这些对象的客户对象提取出来。

参与者与协作者

AbstractFactory为如何创建对象组的每个成员定义接口。一般每个组都由独立的ConcreteFactory进行创建。

效果

这个模式将“使用哪些对象”的规则与“如何使用这些对象”的逻辑分离开来。

实现

定义一个抽象类来指定创建哪些对象。然后为每个组实现一个具体类。可以用表或文件完成同样的任务。

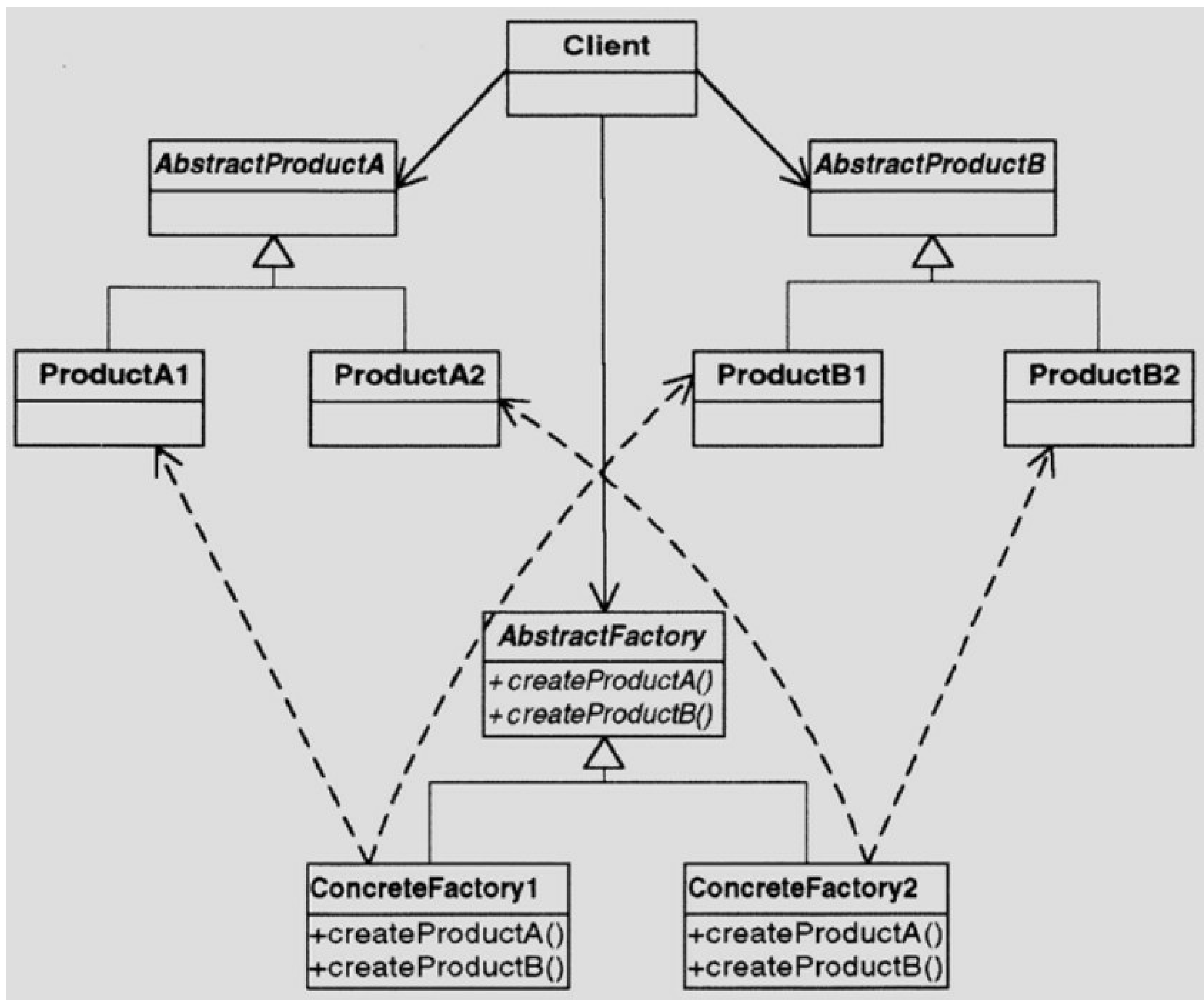


图11-8 Abstract Factory模式的通用结构图

对Client对象隐藏（即封装）了“应该使用哪些服务对象”的选择。这样将来修改选择算法将更加容易，因为不会影响到Client对象。

Abstract Factory 模式为我们提供了一种新的分解方式——根据职责分解。使用这种方法可以将问题分解成：

谁在使用我们的特定对象（ApControl）；

谁来决定使用哪些特定对象（AbstractFactory）。

当有多组对象时，适用 Abstract Factory模式

当问题域中存在不同组的对象，而且每组都用于不同情况时，就应该使用Abstract Factory模式。

你可以根据各种理由来定义对象的组。具体例子如下：

不同的操作系统（编写跨平台应用程序时）；

不同的性能准则；

应用程序的不同版本；

应用程序用户的不同特点；

与地域有关的资源的不同集合（比如方言、日期格式）。

找出对象组和每个组中的成员后，就必须决定怎样实现每种情况（即每个组）。在本例中，通过定义一个抽象类来指定哪些组成员类型可以实例化。对于每个组，再从这个抽象类派生出一个类，实例化组中的成员。

Abstract Factory 模式的一个变体：配置文件

有时候虽然有几组对象，但并不需要每个组都用不同的派生类控制其实例化。可能需要更加动态一些。

例如：

希望用一个配置文件指定使用哪些对象。可以根据配置文件的信息用一个switch语句实例化正确的对象；

可以在数据库中为每个组设一条记录，其中包含使用哪些对象的信息；数据库中的每一列（字段）表示Abstract Factory中每个构造方法使用哪个特定类类型。

进一步的变体：使用Java中的Class类

如果使用Java或者C#，可以将“配置文件”的概念更进一步，用字段表示要用的类名。使用Java的Class类，可以根据这些名字实例化正确的对象。[41]例如，字段可能包含值LRDD表示需要LRDD类的对象。请注意，只要命名规范、定义明确，不需要保存完整的类名。例如，可以在文件中的名字上添加前缀或者后缀。

Adapter模式和Abs-tract Factory

在实际项目中，不同组的成员不会总有共同的父类型。比如，前面

的驱动程序例子中，LRDD 和 HRDD 驱动程序类很可能并不是派生自同一个类。在这种情况下，有必要对它们进行适配，以应用 Abstract Factory模式。

11.6 将Abstract Factory模式与CAD/CAM问题联系起来

在我们的CAD/CAM问题中，系统必须处理许多组的部件，具体情况取决于所使用的CAD/CAM程序的版本。在V1系统中，所有的部件都为V1而实现。同样，在V2系统中，所有的部件都将为V2实现。

因此，将在Abstract Factory模式中使用的组将是V1部件和V2部件。

11.7 小结

本章内容

在必须协调一组对象的创建时，可以应用Abstract Factory模式。它提供了一种方式，将如何执行对象实例化的规则从使用这些对象的客户对象中提取出来。

首先，找出实例化规则，定义一个带接口的抽象类，其中的接口为每种需要实例化的对象提供一个方法。

然后，从这个类为每个组实现具体类。

客户对象使用具体工厂对象创建所需的服务对象。

复习题

简答题

1.虽然使用 switch 语句对于需要在多个方案中进行选择的问题，可能是一个合理的选择，但是它对于本章中讨论的驱动程序问题会导致许多问题。

会有哪些问题？

switch语句的存在说明需要什么？

2.为什么这个模式被称为“Abstract Factory”？

3.Abstract Factory的三个关键策略是什么？

4.此模式中，有两种工厂：

Abstract Factory类的作用是什么？

“具体工厂”类的作用是什么？

5.采用Abstract Factory模式的效果是什么？

阐述题

《设计模式》一书对于Abstract Factory模式的意图是这样描述的：“为创建一组相关或相互依赖的对象提供一个接口，而且无需指定它们的具体类。”

这是什么意思？

给出一个例子。

观点与应用题

1.你认为为什么《设计模式》一书的作者称这个模式为“Abstract Factory”？对于它的功能来说，这是一个合适的名字吗？为什么？

2.你怎么知道何时使用Abstract Factory 模式呢？

---

[1].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979。（中译本《建筑的永恒之道》，知识产权出版社2002年出版。——译者注）

[2].女人类学家Ruth Benedict是基于模式的文化分析的先驱。有关这方面的例子，请参见Benedict R.，The Chrysanthemum and the Sword，Boston:Houghton Mifflin，1946。（中译本《菊与刀——日本文化的类型》，商务印书馆2000年出版。这是一本深刻揭示日本民族性的经典著作，其中题目中类型的英文即pattern。——译者注）

[3].Alexander C.、Ishikawa S.和Silverstein M.，A Pattern Language，New York:Oxford University Press，1977，p.x。（中译本《建筑模式语言》，知识产权出版社。——译者注）

[4].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979。（此段文字摘自该书第6章。由于中译本有些细节过于随意，因而所引文字进行了较大修订。——译者注）

[5].欧洲的ESPRIT协会在20世纪80年代就进行了类似的工作。ESPRIT的1098项目和5248项目发展出一种名为KADS（Knowledge Analysis and Design Support，知识分析与设计支持）的基于模式的设计方法学，主要关注的是有关专家系统构建方面的模式。Katen Gardner将KADS的分析模式扩展到面向对象领域。请参见Gardner K.，Cognitive Patterns:Problem-Solving Frameworks for Object Technology，New York:Cambridge University Press，1998。

[6].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995。（中文版：《设计模式：可复用面向对象软件的基础》，机械工业出版社出版。——译者注）

[7].本部分内容源自Ralph Johnson的一次谈话，作者进行了改编。

[8].请记住，聚集（aggregation）的意思是由互无部属关系的多个事物组成的一个集合（比如飞机场上的飞机），而组合（composition）的意思是某个事物是另一事物的一部分（比如飞机上的轮胎）。这种区分在具有垃圾收集机制的语言中意义不是太大，因为在这些语言中用不着操心对象的销毁（当对象销毁时，它的部分是否也应该销毁？）。（聚集和组合的区别，请参见本书第2章）。

[9].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.139。（中文版：《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，第121页。——译者注）

[10].Facade模式的参与者与协作者是接口本身和各个子系统，此处作者没有明确说明。——译者注

[11].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.139。（中文版《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，第92页。——译者注）

[12].Adapter 模式有两种形式，此处的图对应的是对象 Adapter 模式。另一种类 Adapter模式的情况请参考“实践注记”一节，对应的结构图请参考《设计模式》一书 141页（中文版93页）。——译者注

[13].有关如何在对象Adapter模式和类Adapter模式上进行选择，请参见《设计模式》一书142～144页（中文版第93～94页）。（类Adapter模式的特点是：仅引入一个对象，无需指针间接引用Adaptee；要通过委托一个具体Adaptee类，使Adapter适配接口，因此无法适配所有子类；允许Adapter重定义Adaptee部分行为。对象Adapter模式的特点是：可以适配Adaptee类及其子类；重定义Adaptee比较困难。——译者注）

[14].自开始撰写本书以来，我认识了一些来自Smalltalk社区的人。实际上他们所有人看待面向对象设计的方式都与我在此叙述的类似。

[15].Gamma E.、Helm R.、Johnson R.和Vlissides,J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.29。（中文版：《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，第20页。——译者注）

[16].一般而言，《设计模式》一书中提到“封装”时所指的就是类型封装。

[17].Coplien J.，Multi-Paradigm Design for C++，Boston:Addison-Wesley，1998，p.63。（中文版《C++多范型编程》，中国电力出版社出版。--译者注）

[18].Coplien J.，Multi-Paradigm Design for C++，Boston:Addison-Wesley，1998，p.63。（中文版《C++多范型编程》，中国电力出版社出版。--译者注）p.60，64。

[19].Beck K.，Extreme Programming Explained:Embrace Change，Boston:Addison-Wesley，2000，pp.108－109。（《解析极限编程：拥抱变化》，人民邮电出版社出版。——译者注）

[20].Jeffries R.、Anderson A.和Hendrickson C.，Extreme Programming Installed，Boston:Addison-Wesley，2001，pp.73-74。

[21].“反映意图的名字”就是指能够清晰而且简洁地表达函数要负责完成什么功能的名字。

[22].Fowler M.，Refactoring:Improving the Design of Existing Code，Boston:Addison-Wesley Longman，1999，p.77。（中译本《重构：改善既有代码的设计》，中国电力出版社出版。——译者注）

[23].因为本书讲述的是设计模式，我不会深入讨论预先测试和测试驱动设计这些优秀的实践。

[24].有关测试驱动开发的更多信息，请访问本书的配套网站：http://www.netobjec-tives.com/dpexplained。

[25].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.18。（中文版：《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，第12页。——译者注）

[26].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.20。请注意：OMT（《设计模式》一书所用的建模语言）中的“组合”一词（也为《设计模式》一书所用）在UML（目前标准的建模语言）中称为“聚集”。

[27].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.20。请注意：OMT（《设计模式》一书所用的建模语言）中的“组合”一词（也为《设计模式》一书所用）在UML（目前标准的建模语言）中称为“聚集”。，p.29。

[28].我相信重用并不是使用面向对象方法的原因。降低维护成本和使代码更加灵活（更容易扩展）才是更重要的考虑因素。使用正确的面向对象技术当然可能实现重用，但并不是通过直接使用该对象，然后由它派生新的变体对其重用即可达到的。这样做的结果是产生难以维护的代码。

[29].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.20，29。（中文版：《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，第13页和第20页。——译者注）

[30].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.315。（中文版第208页。——译者注）

[31].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.151。（中文版《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，第100页。——译者注）

[32].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.151。（中文版《设计模式：可复用面向对象软件的基础》，机械工业出版社出版，第100页。——译者注）

[33].Alexander C.、Ishikawa S.和 Silverstein M.，A Pattern Language:Towns/Buildings/Construction，New York:Oxford University Press，1977，p.x。（中英文对照版《建筑模式语言：城镇、建筑、构造》，知识产权出版社。——译者注）

[34].Coplein J.，Multi-Paradigm Design for C++，Boston:Addison-Wesley，1998。（中文版《C++多范型设计》，中国电力出版社出版。——译者注）

[35].“一条规则，一个地方”策略也就是Kent Beck总结的“一次且仅一次”原则。即由（编码人员编写的）代码和测试所构成的系统必须能够表达所应表达的内容，但是不能含有任何重复代码（参见Kent Beck的Extreme Programming Explained。）——译者注

[36].这方面的更多情况，请参阅Coad Peter，Java Design，Upper Saddle River，N.J.:Prentice Hall，2000。

[37].“复合设计模式”曾称为“组合设计模式”，现在的称法是为了与Composite（组合）模式混淆。更多信息，请参考Riehle D.的Composite

Design Patterns 一文，Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA'97)，New York:ACM Press，1997，pp.218-228。另外还可以参考Composite Design Patterns(They Aren’t What You Think)，C++Report，June 1998。

[38].Fowler M.，Refactoring:Improving the Design of Existing Code，Boston:Addison-Wesley，1999，p.xvi。

[39].当然，这里“使用”的意思是“用模式来考虑各种问题、因素等。”

[40].Gamma E、Heml R、Johnson R.和Vissides J.，Design:Patterns Elements of Reusable Object-Oriented Software，Reading Mass.:Addison-Wesley，1995，p.87。

[41].C# 也有等效机制。有关Java Class类的更多信息，请参见Eckel B.，Thinking in Java，Upper Saddle River，N.J.: Prentice Hall，2000。

第四部分 组合起来：用模式思考

概览

本部分内容

在本部分，我将提出一种基于模式设计面向对象系统的方法，该方法已经在我自己的设计实践中得到了检验。我们将运用这种方法解决从第3章以来一直在讨论的CAD/CAM问题。

这种方法首先着力理解对象所处的背景。虽然这种方法在应用中仍有一定局限性，但是它为下一部分要提出的更通用的方法打下了基础。

章 讨论的主题

12 专家设计之道

Christopher Alexander的思想，以及专家们如何使用这些思想进行设计。

13 用模式解决**CAD/CAM**问题

运用这种方法解决第3章第一次提出的CAD/CAM问题。

将这一解决方案与第4章中的方案进行比较。

第12章 专家设计之道

12.1 概览

本章内容

在着手设计时，应该怎样开始呢？先获取各种细节，再看它们如何组合在一起？还是先从总体概念（big picture）开始，再将其逐步分解？或者其他方式？

Christopher Alexander的方法是先把注意力放在高层的关系上——某种意义上也就是自顶向下。他认为在做出所有设计决策之前，理解所要解决的问题的背景是至关重要的。他用模式来定义这些关系。但是，他不仅仅提出了一组模式，还给出了一整套设计方法。他所撰述的领域是建筑学，针对的问题是设计供人们居住和工作的场所，但他的原则也适用于软件设计。

在本章中，我们将：

讨论Alexander的设计方法；

叙述如何将此方法应用于软件领域。

12.2 添加特征的创建方式

《建筑的永恒之道》，一本建筑学图书

现在我们已经理解了几种设计模式，该看看它们怎样协同工作了。对于Alexander来说，只是叙述单个的模式是不够的。他使用这些模式发展出了一种新的设计范型。

……这本书使我成为设计师

他的《建筑的永恒之道》不仅讲述模式，而且讲述这些模式如何协作。这是一部杰作。无论就个人还是专业而言，它都是我最喜欢的书之一。它使我开始欣赏生活中的种种事物，理解自己所处的环境，做出更好的软件设计。

这怎么可能呢？一本讲述建筑设计和城镇规划的书怎么会对软件设计产生如此深远的影响呢？我相信，这是因为它描述了一种范型，一种 Alexander 认为设计师都应遵循的范型。任何设计师皆然。我所感兴趣的正是这种设计范型。

我多么希望自己能这样说："我一读到这本书，就立刻接受了Alexander的观点。"但是，事实并非如此。我对这本书的第一反应是："这很有意思。很有道理。"然后我又回到了老路上去，毕竟老方法我已经用了很久了。

还是老话说得好："机遇总是青睐有准备的人"。好在我很幸运，于是一切不再相同。

在阅读《建筑的永恒之道》几个星期后，机遇降临了。在一个项目的设计中，我的标准方法无能为力。做出的几个设计都不够好，所有曾经经过检验的设计方法都失败了，我非常沮丧。幸运的是，我还算聪明，想到可以尝试一种新的方法——Alexander的方法，结果让人喜出望外。

"将适合的事物组合起来"进行建设

我们将在下一章描述具体的过程。我们先来看看 Alexander 是怎样教导我们的。

设计常常被认为是一种合成过程，一种将事物放在一起的过程，一种组合过程。按照这种观点，整体是由部分组合而成的。先有部分，然后才有整体的形式。[1]

从部分到整体、从已知的具体事物开始设计，这很自然。

当我第一次读到书中这段话时，我想："是啊。这与我观察事物的

方式非常类似，先确定需要些什么，然后将它们组合起来。”也就是说，我先找出类，然后观察它们如何协作。组合这些部分之后，可以回头来看看它们是否适合总体概念。然而，即使在将注意力从局部转到全局时，整个过程中我考虑的仍然是这些部分。

对于面向对象开发人员而言，这些部分就是对象和类。我找出对象和类，为它们定义行为和接口。但由于我是从这些部分开始的，注意力也总是放在这些部分上。

但是，这可能不是好思路

回想一下第4章中CAD/CAM问题最初的解决方案。开始想到的是需要许多不同的类：沟槽、孔、方切口等等。因为已经知道需要将这些类与V1系统和V2系统关联起来，所以我认为需要一组使用V1的类和一组使用V2的类。给出这些类之后，再来看如何将它们联系起来。

但是，只是把预先形成的部分加起来，不可能形成有自然特征的任何东西。[2]Alexander认为，从部分构造看并非好的设计方式。

尽管Alexander讨论的是建筑学，但我尊敬的许多软件设计人员都说，他的真知灼见对于我们同样适用。看来我必须虚心地接受这种新的思考方式。好吧，我试试看，于是我听见Alexander好像在说：“将预先形成的部分加起来是无法获得优秀的软件设计的。”在软件行业，所谓“将预先形成的部分加起来”，就是说先创建一个例程库，然后再将它们组合起来，构建出系统。

从部分构造整体，不可能得到优美的设计

当各个部分都是模块化的、而且在形成整体之前已经产生时，它们按照定义应该完全相同。因而每个部分不可能只根据自己在整体中的位置而具有独特性。更为重要的是，这种模块化的部分所形成的任何组合根本不可能包含大量同时出现在有生气的建筑中的模式。[3]

一开始我对Alexander关于模块化的论述并不理解。按道理要实现重用，应该必须给出一些通用的例程，但是他却在说不要这样做。后来，

我认识到，所谓“模块化”，Alexander是指在建筑业中那些完全相同、可以互换的部分，他不是指我们软件行业中所用的术语“模块”。他的意思是，如果在形成总体概念之前就开始构建模块，这些模块不可能有能力处理特殊的需要。

似乎在 Alexander 的方法和重用目标之间我们陷入了进退两难的境地。不用担心，本书最后会解决这一难题。我希望对 Alexander 论述的理解能够像当初对我自己那样对你有所帮助。遵循了 Alexander 的方法之后，我发现自己创建的类比从通用部分出发所得到的更好，而且重用性也更强。

优秀的设计要求胸中始终有丘壑

每一部分都根据自己在整体中的位置而改变，只有通过这样的过程，一个建筑才能富于生气。[4]

在读到“有生气”这个词的时候，可以将它等同于“健壮而灵活的系统”。

在此之前，Alexander 曾提到，各部分需要有其独特性，从而可以利用各自所处的具体环境。现在，他将这一论述更深一步。正是在与周围环境的对抗和适应中，建筑物获得了自己的特征。考虑一下这些建筑学方面的例子。

瑞士村庄——你的眼前浮现出一个村庄，村舍紧密依偎，彼此非常相似，但是每座村舍又各有特点。它们之间的差异并非毫无规律可寻，而是反映了建造者和房主的经济考虑，以及房屋与周围环境融合的要求。其结果是构成了一幅优美、舒适的图画。

美国城郊居民区——所有房屋的设计都千篇一律。几乎没有考虑房屋周围的自然环境。所有契约和标准都是为了保证这种同质性。最终房屋没有了个性、毫无可爱之处。

现在将这些应用于软件设计似乎太“概念化”了。我们的目标是在它们所处的大背景中设计部分——类和对象，从而构造出健壮而灵活的系

统，这里只需理解这一点就够了。当然，问题在于："怎样才能做到这一点呢？"

包含复杂化的设计过程

一言以蔽之，每个部分都因其存在于更大整体的背景中而被赋予了特定的形式。

这是一个分化的过程。它把设计看成是一系列复杂化（comple-xification）的活动；结构是通过对整体操作、使其起皱（crinkling it）而注入其中的，而不是通过一小部分一小部分添加而成。在分化的过程中，整体孕育了部分：整体的形式及其各个部分是同时产生的。分化过程的好像是胚胎的成长过程。[5]

"复杂化"。这个词到底是什么意思呢？难道我们的目标不是让事情更简单，而非更复杂吗？

Alexander所描述的是这样一种设计思想方式，开始用最简单的术语来考虑问题（概念性层次），然后添加更多特性，在此过程中设计将越来越复杂，因为我们加入了越来越多的信息。[6]

这是一种非常自然的过程。我们其实总是这样做的。例如，假设需要为一个有 40 名听众的演讲安排房间。当你向别人描述需求时，可能会这样说："我需要一间30英尺长30英尺宽的房间"（开始很简单）。然后是："我想把椅子按剧院的方式安排：4行，每行8把"（添加信息，使房间的描述更加复杂）。接下来是："房间的前面需要放一个讲台"（更加复杂）。

我们是怎么做的？设计的过程如何？

在语言的影响下，创造者心中设计的展开过程正与此相同。

每一个模式都是一个分化空间的操作符，也就是说，它在以前没有差异的地方创造差异。

而在语言中，操作是按顺序安排的，因此像已经进行的那样，操作一个接一个，逐渐地产生出一个完整的东西。从与其他相似的东西共享

模式的意义上而言，语言具有一般性；从它在其具体环境唯一的意义上而言，语言又是特殊的。

语言就是一系列这样的操作符，其中每一个操作符进一步对以前分化形成的意象继续分化。[7]

Alexander 断言：设计应该从问题的一个简单陈述开始，然后通过在这个陈述中加入信息，使它更加详细（也更加复杂）。这种信息应该采取模式的形式。对于Alexander而言，模式定义了问题域中实体之间的关系。

例如，我们来考虑第5章中讨论的庭院模式，该模式必须描述庭院中所含实体以及它们之间的关系，这些实体包括：

庭院的开放空间；

穿过庭院的小径；

外眺的景色；

甚至要使用这个庭院的人们。

通过思考实体互相之间的关系应该怎样，为我们设计庭院提供了大量信息。再思考庭院模式的背景中存在的其他模式，比如门廊或朝向庭院的阳台，可以改进庭院的设计。

使这种分析方法如此富于成效的原因，是它不必依赖我的经验、直觉或者创造力。A1exander 认为，这些模式的存在独立于任何人。一个空间富于生气，是因为它遵循了自然的过程，而不仅仅因为设计师是个天才。因为设计的质量取决于是否遵循这种自然的过程，所以看到类似问题的高质量的解决方案都非常相似，也就没有必要感到惊讶了。

以此为基础，他还指出了优秀设计师应该遵循的规则。

每次一个——模式应该按顺序每次只运用一个。

背景优先——首先应用那些要为其他模式创造背景的模式。

模式定义了关系

Alexander 所描述的模式定义了问题域中实体间的关系。这些模式的重要性次于这些关系，但它们为我们提供了一种讨论这些关系的方式。

后续步骤

Alexander 的方法也同样适用于软件设计。也许不能原样照搬，但基本原理肯定是适用的。Alexander会对软件设计师说些什么呢？

Alexander 的步骤	讨　论
找出模式	找到问题中存在的模式，用这些模式来思考问题。请记住，模式的用途是定义实体之间的关系
从背景模式开始	找出为其他模式创造了背景的模式。这些模式应该作为设计的起点
然后，从背景转向内部	观察其余的模式和任何其他可能已经发现的模式，从中选出为其余模式定义背景的模式。重复这一过程
改进设计	改进过程中始终考虑模式所蕴涵的背景
实现	实现应该融入模式所要求的细节

这能够奏效吗？

这能够奏效吗？有时候能。这种方法的问题在于，它假设你能够发现问题中的已知模式。有时候确实如此，但是并不经常。当然，不需要任何模式预备知识的更通用的方法是存在的。只不过在叙述这种方式之前，我要先举一个前面所述的 Alexander 方法能够奏效的例子：CAD/CAM问题。

在软件设计中使用**Alexander**的方法：个人观点

第一次使用Alexander方法时，我非常教条地从字面上理解他的话。他的概念来源于建筑学，通常不能直接转换到软件设计（或其他类型的设计）中。从某些方面而言，我为自己使用设计模式的早期经验感到庆幸，因为我要解决的问题中模式是以定义非常明确的背景顺序出现的。但是，这也有不利的一面，因为我因此天真地认为这种方法一般都能奏

效（其实并非如此）。与此同时，软件界的许多重要的设计师都在提倡开发“模式语言”——寻求将Alexander的方法应用于软件开发的正规方式。我将这些理解成了差不多可以将Alexander的方法直接应用于软件设计。（现在我当然不再这么想了。）因为Alexander说建筑学中的模式有预定的背景顺序，所以我曾经认为软件模式也有这种预定顺序。也就是说，一类模式总是会为另一类模式创造背景。我开始在给别人讲课时宣讲我自己理解的Alexander方法。几个月、几个项目之后，我开始看到问题了。有些情况下背景的预定顺序无法不起作用。换句话来说，就是有时候Bridge模式会为Composite模式创造背景，而有时候恰恰相反。

由于有过数学专业方面的训练，只需一个反例就足以证明我的理论有误。这使我开始怀疑自己方法中的一切——而这是我本来经常做却因为激动而忘掉的事情。

自从这一早期阶段之后，我懂得了Alexander的工作中那些原则更加重要。尽管在建筑学和软件开发中表现方式不同，但这些原则确实都适用于软件设计。我在设计的改进中看到了这一点。我在更快、更健壮的分析中看到了这一点。每次维护自己的软件时，我都能体验到这一点。

12.3 小结

本章内容

设计常常被认为是一种合成过程，一种将事物放在一起的过程。在软件中，常用的方法是立即寻找对象、类和组件，然后思考它们如何互相配合。

在《建筑的永恒之道》一书中，Christopher Alexander 描述了一种更好的方法，一种基于模式的方法。

1.从对整体的概念性理解开始，以理解需要实现的目标。

2.找到在整体中出现的模式。

3.从为其他模式创造背景的那些模式开始。

4.从背景向内：应用这些模式，找到新模式，并重复。

5.最后，通过每次应用一个模式，改进设计，并在所创建的背景中予以实现。

作为一个软件开发人员，你可能无法直接应用A1exander的模式语言方法。但是，通过在已经出现的概念的背景中添加新概念进行设计，肯定是我们每个人都可以做到的。学习本书后面的新模式时，请牢记这一点。许多模式都能够创建健壮的软件，因为它们定义了实现类能够在其中工作的背景。

复习题

简答题

1.Alexander使用“有生气”这个术语描述优秀的设计。对于软件书中建议使用什么术语？

2.优秀的设计需要将什么牢记在心？

3.Alexander建议最好的设计方法是“复杂化”。这是什么意思呢？

4.对于Alexander而言，模式定义了什么关系？

5.Alexander的五步设计法是什么？

阐述题

Alexander说：“只是把预先形成的部分加起来，不可能形成有自然特征的任何东西。”他这是什么意思？

观点与应用题

1.有时候，在面向对象程序设计中，会提供许多小的、可重用组件，可以将它们组合起来以创建一个程序。这与Alexander的理论一致还是相悖？或者Alexander的说法是在不同层次上？说明理由。

2.你见过某个建筑中有让人感觉极为“死气沉沉”或者令人不舒服的庭院或者入口通道吗？按照Alexander对庭院模式的描述，你所见的庭院或者入口通道没有解决或者包含哪些实体？

3.举出一个你认为Alexander的方法可以适用的软件项目，再举出一个不适用的。问题何在呢？在阅读本书此后的章节时，请将这一案例牢记在心。

第13章 用模式解决CAD/CAM问题

13.1 概览

本章内容

本章中，我将运用Alexander所建议的设计模式方法[8]来解决第3章中提出的CAD/CAM问题。

在本章中，我们将：

循序渐进地讲述解决CAD/CAM问题所需的方法；

引导读者经历初期设计阶段，而实现细节留待读者完成；

比较新的解决方案与此前的解决方案。

13.2 对CAD/CAM问题的回顾

需求

第3章中描述了CAD/CAM问题的需求，正是这个来自实践的问题使我转向了使用设计模式的道路。

问题域是支持一个大型工程公司的计算机系统，更具体地说来，是支持这个公司的CAD/CAM系统。

基本需求是：创建一个读取CAD/CAM模型并从中提取部件的计算机程序，将这些部件提供给一个现成的专家系统，从而进行智能化设计。该程序使专家系统能够无需考虑CAD/CAM系统的具体细节。复杂之处在于，CAD/CAM 系统还在修改之中，有可能专家系统必须与多个版本的CAD/CAM系统交互。

在与用户初步会谈之后，我总结出图 13-1 所示的高层的系统架

构，还有系统的如下表中所示的需求。

需　求	描　述
读取 CAD/CAM 模型并提取部件	系统必须能够分析和提取金属板材各部分的 CAD/CAM 描述 然后，专家系统确定如何制造金属板材，生成所需指令，由机器人将板材制造出来
能够处理多种零件	最初关注的是金属板材零件 每个金属板材零件可能有多种特征，包括沟槽、孔、方切口、特殊形状和不规则形状部件。将来不可能出现其他的部件
处理多个版本的 CAD/ CAM 系统	从图 13-1 可以推出：需要能够在不修改专家系统的情况下，即插即用不同的 CAD/CAM 系统

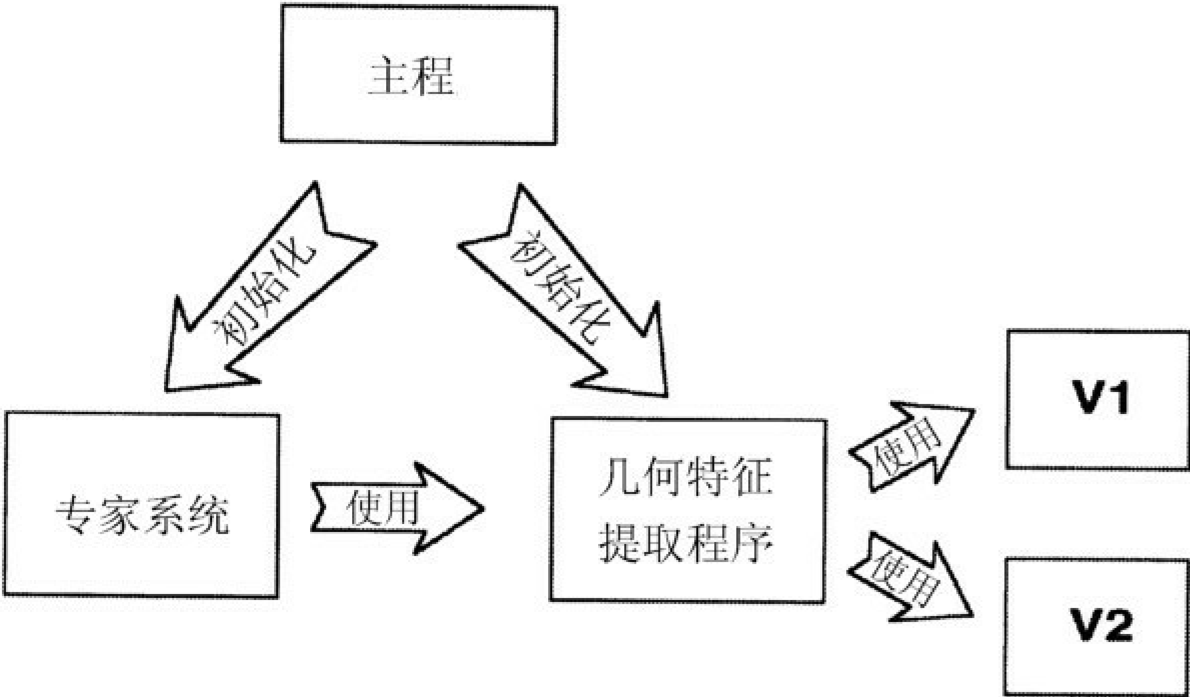


图13-1 解决方案的高层视图

13.3 用模式思考

用模式思考的步骤

我着手处理一个设计问题时，并不总是去尝试“用模式思考”。但是经验证明，用模式思考肯定有助于获得突破性的思路。模式为我的思考指引了方向。它引导我形成了自己的行之有效而且能够推而广之的方

法，本书稍后将讲述这一方法。

我开始领悟到 Alexander 的方法，是在花费了许多时间为了给CAD/CAM问题找到比第4章“标准的面向对象解决方案”中给出的方案更好的解决方案，但是最终徒劳无益之后。当时直觉告诉我，有更好的解决方案，但是我苦思冥想而不得。

我决定休息一下，出去走一走，清醒一下头脑，努力不再去想这个问题。令人惊奇的是，刚刚走出 30 步，我就如醍醐灌顶，顿悟了。再次开始时，我已经改弦更张。我想象自己在和Alexander交谈。我问大师：“如果是你，会怎么做呢？”我能够听到他这样回答：“用模式进行思考！”我好像还听到他微笑着，加了一句：“小傻瓜！”

回想前面一章中所述的各种概念，我突然明白他所指为何——用在问题域中发现的那些模式进行思考！只考虑关键的概念，先不要操心细节。

我曾经这样非常正式地表述用模式思考的过程。

用模式思考的过程

1.找出模式。在问题领域中找出模式。

2.分析和应用模式。对于要进行分析的模式集合，执行步骤2a～2d。

2a.按背景的创造顺序将模式排序。根据为其他模式创造背景的情况将模式排序。其原理是，一个模式将为另一个模式创造背景，不会出现两个模式互为彼此创建背景的情况。

2b.选择模式并扩展设计。根据排序，选择列表中的下一个模式，用它得到高层的概念设计。

2c.找到其他模式。找到在分析中可能出现的其他模式，将它们添加到要分析的模式集合中。

2d.重复。对还没有融入概念设计的模式重复以上步骤。

3.添加细节。根据设计的需要添加细节。扩展方法和类的定义。

必须承认，只有能够用模式来理解整个问题域时，这种方法才可以发挥作用。而且虽然它并非总能奏效，但是它为我们提供了很好的起点。设计模式最有用的地方，可能就是提供了着手的方法。此后还需要通过找出问题域中各种概念之间的关系，充实其他部分。这就需要使用共性/可变性分析（CVA）方法，我们将在第15章中讨论。共性/可变性分析是普适的，这一点与“用模式思考”不同。我们之所以从“用模式思考”开始，是因为它更容易学习，而且有助于理解更为通用的共性/可变性分析方法。

我们接下来逐一讨论CAD/CAM问题中所用的用模式思考的各个步骤。

13.4 用模式思考：步骤1

1.识别模式

前面的章节中，已经从CAD/CAM问题中找到了如下4个模式：

Abstract Factory

Adapter

Bridge

Facade

到目前为止还没有出现其他模式，但我们对可能出现的更多模式是敞开大门的。

注意：如果是初次接触这一问题，我会采用前面所提出的思考过程：约束因素有哪些？问题域中的问题何在？它们与我知道的模式如何对应？等等。

13.5 用模式思考：步骤2a

按背景考察模式

下面将逐一考察已经在问题域中找出的模式。然后根据每个模式如何为其他模式创造背景进行选择。

2a.看哪个模式为其他模式创造背景

确定问题域中哪些模式为其他模式创造背景时，我运用了一种简单的技术：考察所有模式的可能配对，每次考察两个。这里存在6种可能配对，如图13-2所示。

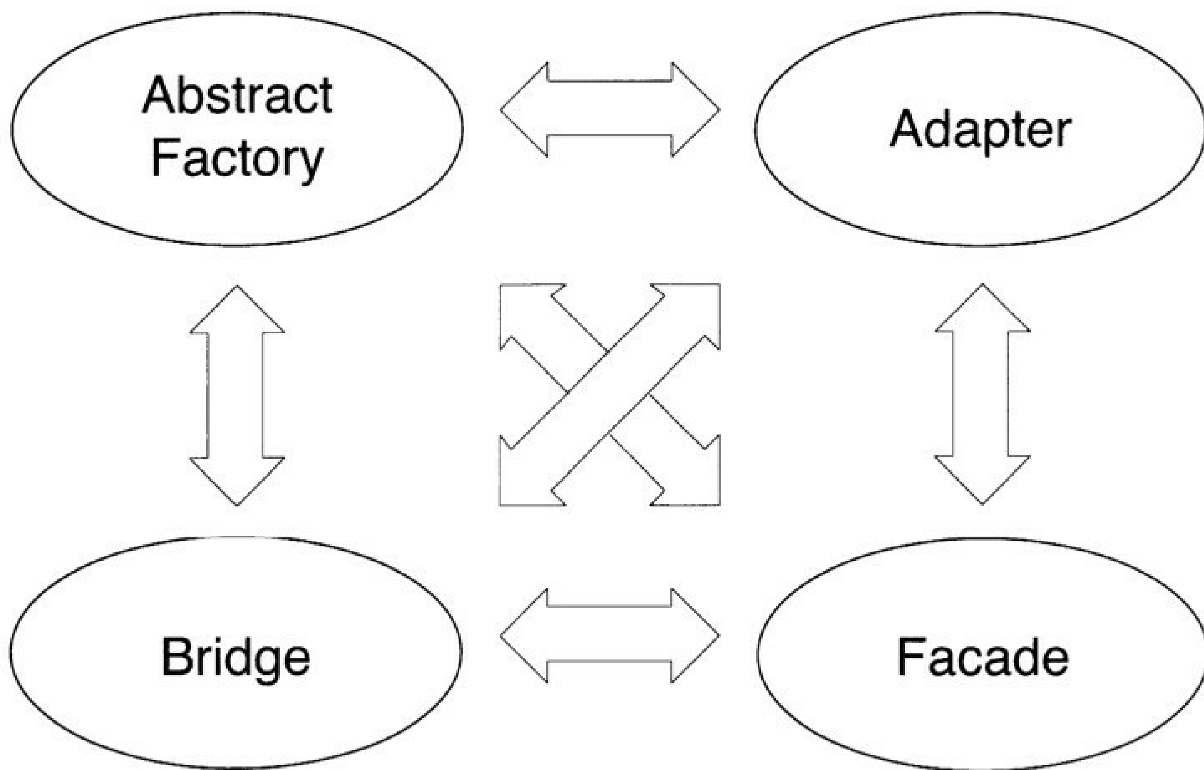


图13-2 模式之间可能存在的各种关系

这个过程不会耗费太长时间

如果还有其他几个模式，这一过程似乎会变得非常棘手。情况并非如此。如果有一些经验，可以很容易地事先将许多模式从主要模式的竞争中淘汰出去。无论如何，通常总是只需要处理几个模式。

在这里，通过几种组合，就完全足够考察所有可能性。

寻找什么在创造背景

我们所说的“一个模式为其他模式创造背景”到底是什么意思呢？“背景”的一种定义是：一些事物存在或发生所处的相互有关的情况——一种环境、一种场景。

在第5章的庭院例子中，Alexander 指出一个门廊存在于庭院的背景中。庭院定义了门廊存在的环境，或称背景。

系统中的一个模式经常与其他模式相关——它为系统中的这些其他模式提供背景。在分析中，寻找一个模式与其他模式是否相关、如何相关，寻找这个模式为其他模式创造或提供的背景，以及这个模式本身存在的背景，都是非常重要的。也许这些并非每次都能找到。但是，通过寻找，你将得到更高质量的解决方案。

寻找背景是一种非常基本的工具，应该加入你的分析和设计工具箱中。确定哪个模式会为哪个模式创建背景，经常要借助于对模式的概念性考察。例如：

Abstract Factory 模式创造了相关对象（族）的集合；

Adapter模式使已有的类A能够适配使用类（using class）B所需的接口；

Bridge模式允许一组相关的使用对象（using object，此模式中抽象的具体类）使用不同实现；

Facade模式能够为使用类B简化已有系统A。

使用这些概念

考虑两个模式时，可以使用模式的这种“元”描述，分析哪个模式为另一个模式创造背景。虽然我可以立即将Abstract Factory排除在列表中第一模式之外（如果已经读过了下面的旁注的话），但是还可以通过其元描述发现，其背景是通过组织要创建的对象（族）而定义的。所以，肯定有其他模式创造了它的背景，它不可能是列表中的第一个。

事实上，Abstract Factory是列表中最后一个模式（除非在最初的设计期间又有其他创建型模式出现，如果是那样的话，这些创建型模式将竞争末位）。

剩下3对

现在，还有3对模式需要考虑：

Adapter-Bridge

Bridge-Facade

Facade-Adapter

考虑背景时使用的一条规则

在一个项目中，我对自己的设计方法进行了反思。我注意到自己一直在无意识地这样做：在知道所需对象是什么之前，并不考虑如何实例化这些对象的问题。我主要关注的是对象之间的关系，似乎对象已经存在。我假设在适当的时候，能够构造出适合这些关系的对象。

这样做的原因在于：在设计中需要尽量减少脑子里思考的事情。以后再考虑如何实例化适合需求的对象通常不会有太大的风险。过早考虑反而会起副作用。在知道需要实例化什么对象之前，不考虑对象实例化更好。明天的事明天说——至少对于实例化是如此！

也许有读者会觉得这样做很有道理。我还没有听说有人将此作为一条规则的，因此在采纳为一条通用规则之前，需要检验一下。我相信自己作为一位设计师的直觉，但是我当然远未达到尽善尽美。所以，我与其他一些经验丰富的开发人员对此进行了交流；他们无一例外，都遵循着这一规则。这给了我足够的信心，将这条规则告诉广大读者。

规则：先考虑系统中需要什么，然后再去关注如何创建它们。

在按Alexander的方法考虑背景时，这条规则就意味着使用对象所引出的模式将为实例化对象的模式（经常归类为创建模式）创造背景。也

就是说，我们应该在确定了对象是什么之后再定义工厂。[9]

最高模式将约束其他模式

所谓“最高模式”是指系统中为其他模式建立背景的一两个模式。这个模式将约束其他模式的行为。你还可以使用的其他术语包括“最外层模式”或“背景设定模式”。

与新接触模式的人一样，我也可能看不出哪个模式明显地依赖于其他模式，或者哪个模式能够为其他所有模式设定背景。

在没有明显的选项时，必须按部就班地逐一在模式组合中寻找如下问题的答案。

有没有一个模式定义了另一个模式的行为?

有没有两个模式彼此相互影响?

Adapter 模式的含义就是“将一个类的接口转换成客户希望的另外一个接口”。在这里，需要适配的接口是OOGFeature。Bridge模式的含义是“将一个抽象的多个具体实例与其实现分离”。在这里，抽象是Feature，实现是 V1 和 V2 系统。听起来似乎 Bridge 模式需要 Adapter 模式来修改OOGFeature的接口，也就是说，Bridge模式需要使用Adapter 模式。

模式之间有关系吗？

显然，Bridge模式和Adapter模式之间存在某种关系。

模式是相互联系的吗？

我可以在没有另一个模式的情况下定义一个模式吗，或者模式中的一个需要另一个吗?

通过考察模式，我们会知道该怎么做。

无需确切知道如何使用V1和V2系统，我就可以说，Bridge模式将

Feature与V1、V2系统分离了。事实上，使用Bridge的一个标准方式就是定义抽象的实现，使抽象在无需关注具体实现的不同情况下使用它。然后，在定义了抽象的实现之后，具体实现将进行适配，从而能够从抽象实现类中派生。

但是，如果不知道要修改成什么样，我就不能说“用Adapter模式来修改V2系统的接口”。如果没有Bridge模式，接口就根本不存在。Adapter模式的作用正是为了将V2系统的接口修改成Bridge模式定义的实现接口。

因此，Bridge模式为Adapter模式创造了背景。我可以把Adapter模式也从最高模式候选中排除掉了。

背景和被使用之间的关系

通常情况下，当一个模式使用另一个模式时，被使用的模式就处于使用模式的背景中，这条规则也可能有例外，但是大多数时候它都是成立的。

去掉一个，还有两个

现在我只需要比较Bridge-Facade和Facade-Adapter了。

我将首先考察Bridge模式和Facade模式之间的关系，因为如果Bridge模式是最高模式，就不需要再考虑Adapter-Facade关系了。（请记住，现在只需要找到最高模式。）

Bridge-Facade关系

适用于Bridge和Adapter的逻辑，很显然也同样适用于Bridge和Facade。

我将使用Facade模式来简化V1系统的接口。

但是谁要使用我创建的新接口呢？Bridge模式的一个实现。

胜利者是 Bridge 模式

因此，Bridge 模式为 Facade 模式创造了背景。Bridge 模式就是最高的模式。

按照Alexander的说法，应该从整体开始。可是回到起点时，我发现还没有Bridge模式的背景呢。

13.6 用模式思考：步骤2b

问题作为整体进行叙述

所以，我回溯设计步骤，找到出现Bridge模式的背景。我希望建造一个系统，将CAD/CAM模型翻译成NC集（制造零件的指令），并将其提交给机器，制造出模型所描述的零件（见图13-3）。



图13-3 系统的高层视图

扩展了的设计

当然，我注意到可以使用面向对象设计技术，让专家系统使用一个Model类获取所需的信息，因此已经对设计进行了扩展。Model类应该有两个版本，每个版本对应一个CAD/CAM系统版本，如图13-4所示。

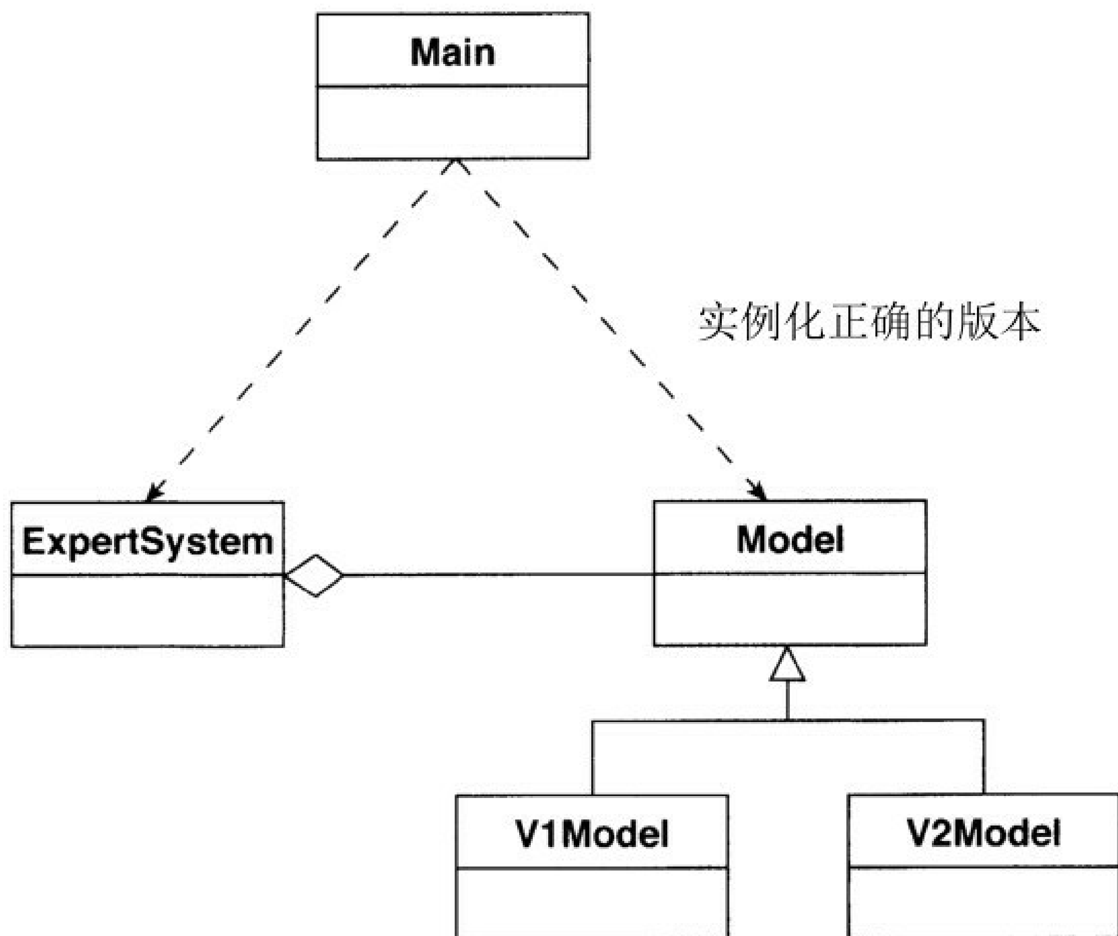


图13-4 系统的高层视图

进一步扩展设计

请记住，我无需关心专家系统的设计。尽管它本身很有意思（而且很多方面而言它更具挑战性），但其设计已经完成了。应该集中精力进行Model类的设计。我知道Model由许多Feature组成，如图 13-5 所示。[10]

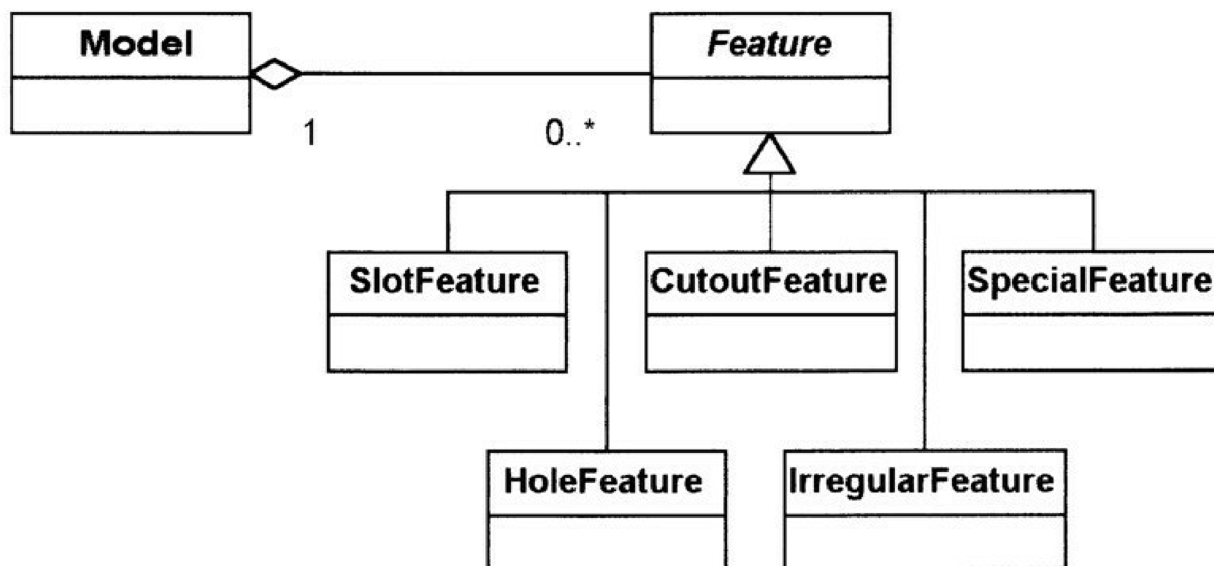


图13-5 Model类的设计

为Bridge模式做好准备

现在，Bridge 模式已经万事俱备了。显然会有多个 Feature（抽象）和多个CAD/CAM系统（实现）。这些就是为Bridge模式设置背景的对象。

如何处理Model类?

Bridge 模式将 Feature 与不同的 CAD/CAM 系统实现联系起来。Feature 类就是 Bridge 模式中的 Abstraction，而 V1、V2 系统就是implementation。但是Model类如何呢？这里也存在Bridge模式吗？并非如此。我可以用继承来创建Model，因为 Model类唯一变化的地方就是被使用的实现。在本例中，可以为每个CAD/CAM系统创建一个Model类的派生，如图13-6所示。如果尝试为Model类使用Bridge模式，那么将得到图13-7所示的设计。

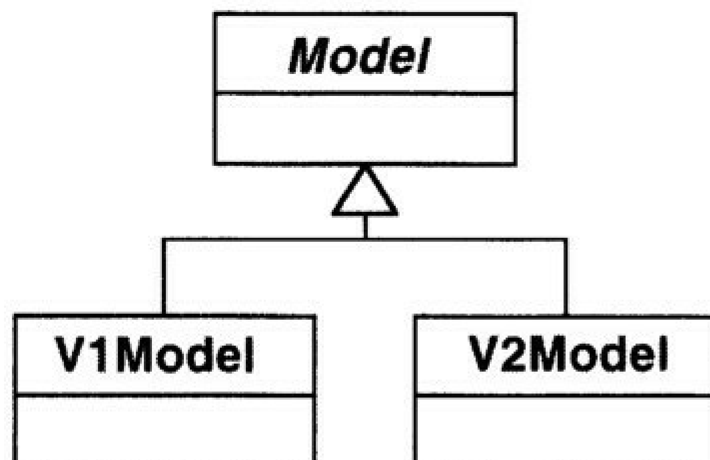


图13-6 使用继承来处理两种模型

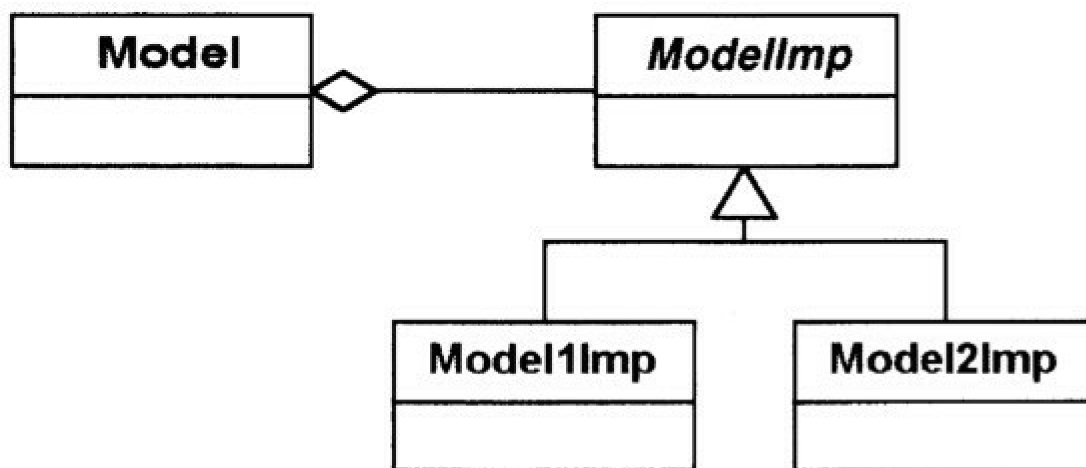


图13-7 使用Bridge模式来处理两种模型

请注意，图13-7中并不是真正的Bridge模式，因为除了实现之外Model类没有其他变化。最多也只能说是一种退化了的Bridge模式，因为只有一个具体的Model。而在Feature类中，有不同类型的Feature，它们拥有不同类型的实现——这里的确存在Bridge模式。

从一般形式开始

将Feature类作为抽象，将V1和V2作为实现的基础，我们来开始实现Bridge模式。为了将问题转换为Bridge模式，我从Bridge模式的标准形式开始，然后替换其中的类。图 13-8 所示为该模式的一般形式（有时候也称为规范形式）。

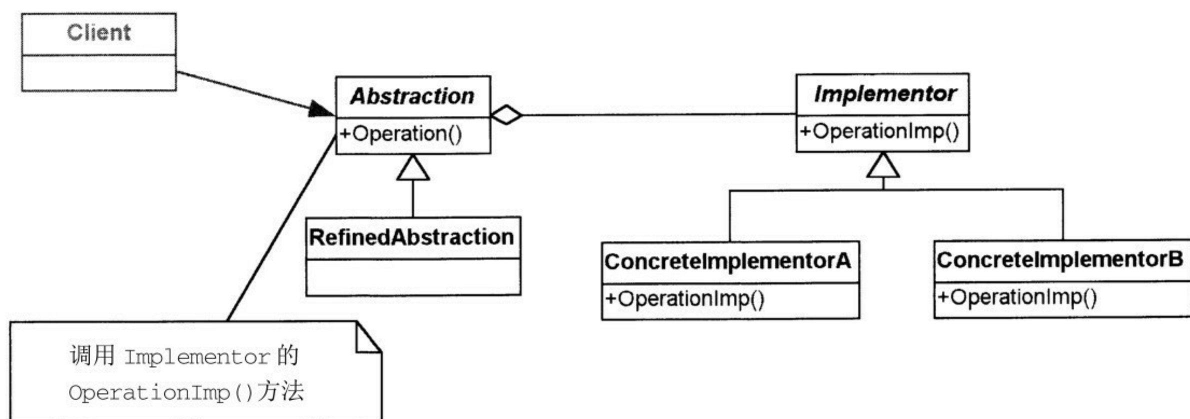


图13-8 Bridge模式的一般结构

……然后对应类

在这个问题中，Feature对应于Abstraction。存在 5种不同类型的部件：沟槽、孔、方切口、不规则形状和特殊形状。V1和V2系统是实现；我选择将负责这些实现的类分别命名为V1Imp和V2Imp。

将类替换进标准的Bridge模式，得到图13-9。

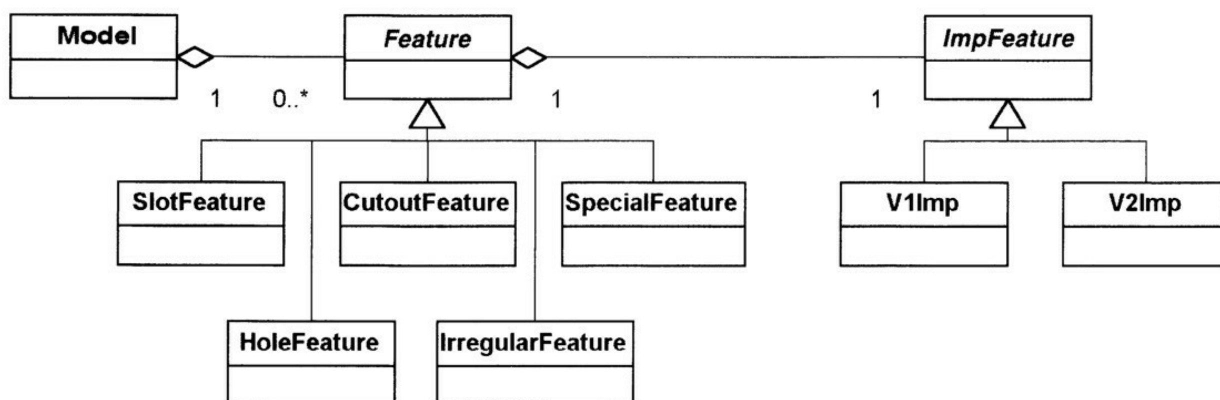


图13-9 将Bridge模式应用于本问题

在图13-9中，Feature将由 ImpFeature——V1Imp或 V2Imp来实现。在此设计中，ImpFeature必须有一个接口，允许Feature获得所需的信息并提供给 Model。因此，ImpFeature的接口应该包括诸如下列的方法。

getX：获取Feature的x坐标。

getY：获取Feature的y坐标。

getLength：获取Feature的长度。

它还需要有只由某些Feature使用的方法：

getEdgeType：获取Feature边缘的类型。

注意：只有需要这一信息的部件才会调用此方法。后面我们将讨论如何使用这些背景信息辅助代码的调试。

13.7 用模式思考：步骤2c

革命尚未成功

我可能还无法看出如何完成实现，不过没有关系。还可以应用其他模式。

寻找其他模式

观察图13-9，应该自问：是否还有其他模式前面没有发现？看不出还有什么其他模式了。唯一的问题是将V1、V2　CAD/CAM系统加入这个设计。这正是Facade模式和Adapter模式的用武之地。

模式有助于我们看到可能性

使用模式还有另一个优点：可以看到重用和灵活性的可能，这是仅仅陷于细节的时候根本无法看到的。例如，使用Bridge模式，我能够注意到在某种层次上V1和V2系统现在是可以互换的——至少对于使用它们的部件对象而言是相同的。在前面比较糟糕的设计中，可能也是如此，但是我当时还看不到这一点。这种灵活性被细节遮住了。模式方法有助于我们更清晰地看到这种可能性。

13.8 用模式思考：重复步骤2a和步骤2b（Facade模式）

其他模式能否为别的模式创造背景？

接下来，我要检查其他模式是否会为别的模式创造背景。在本例中，Facade模式和Adapter模式显然与设计的不同部分存在关联，它们彼此无关。因此，我可以选择任何顺序应用它们。任意选择好了，下一步

应用Facade模式，得到图13-10。

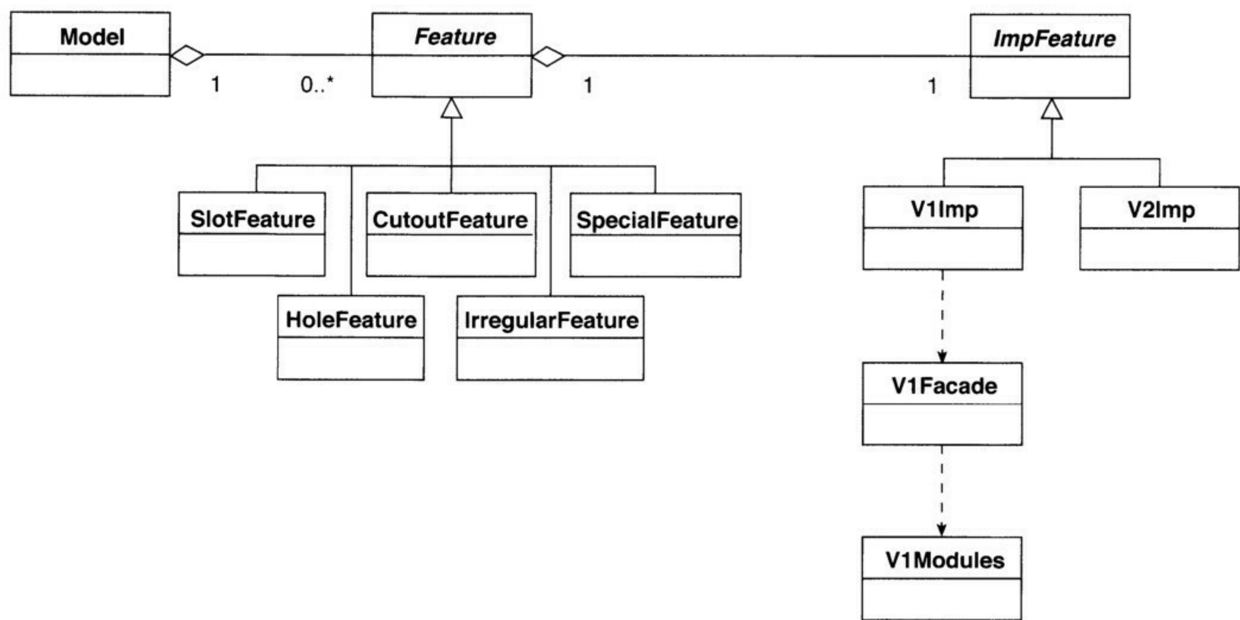


图13-10 应用Bridge模式和Facade模式之后

Facade模式

应用Facade模式意味着在V1模块和使用它们的V1Imp对象之间插入一个Facade对象。V1Facade有一些简化的方法，与V1Imp需要完成的操作对应。V1Facade中的每个方法看起来都像是对 V1 系统的一系列函数调用。

调用这些函数所需信息的种类将决定 V1Imp 如何实现。例如，使用Vl系统时，需要告诉它使用哪个模型、Feature的ID是什么，因此，所有使用V1Facade的V1Imp对象都需要知道这一信息。因为这是特定于实现的信息，所以 VlImp需要自己知道，而不是通过调用 Feature获取。这样，在V1系统中，每个Feature都需要有自己的V1Imp对象（以保存特定于系统的部件信息）。在一般架构完成后，我们再来更详细地讨论这一点。

利用特殊性来调试代码

在本章的前面部分，我提到过：实现部分的某些方法应该只由某些Feature对象调用。我可以利用对“什么应该调用什么”的了解在代码中设置检查点。我并不需要这样做，如果规则改变我可能需要移除这些检查点。然而，一开始这可能是有用的方法。

例如，在这个 CAD/CAM 解决方案中，Feature对象包含一个实现部分对象。实现部分的方法之一是 getEdgeType。只有当 Feature对象是一个沟槽或方切口时，这个方法才有意义。其他的Feature对象没有“边缘类型”的概念。如果我正确地实现了所有东西，getEdgeType方法永远不会被沟槽和方切口之外的任何 Feature 对象调用。只要在getEdgeType方法中加入一条断言来检验作出调用的Feature对象的类型是否合适，我就可以检查是否有不正确的调用发生。

13.9 用模式思考：重复步骤2a和步骤2b（Adapter模式）

下面是Adapter模式

应用了Facade模式之后，现在来应用Adapter模式，得到图13-11。

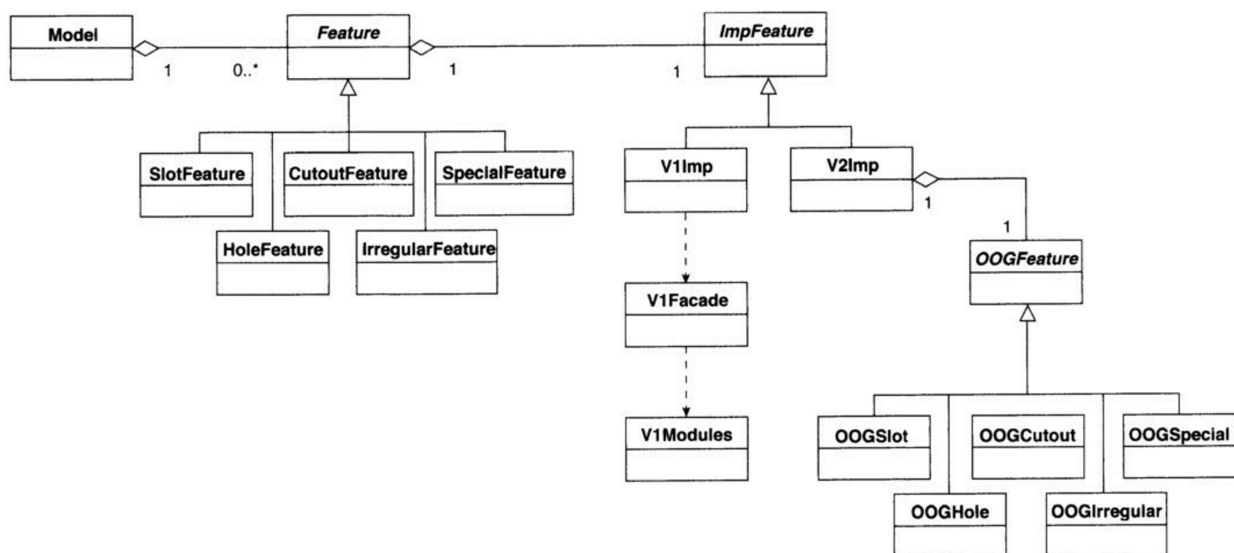


图13-11 应用Bridge模式、Facade模式和Adapter模式之后

13.10 用模式思考：重复步骤2a和步骤2b（Abstract Factory模式）

剩下的只有Abstract Factory模式了。实际上，这个模式其实并不需要。使用Abstract Factory模式的根本理由在于：如果有一个V1系统或者V2系统，需要确保所有的实现对象都是V1类型或者V2类型的，但是，Model对象本身已经知道这一点。如果其他对象可以很容易地封装创建规则，就没有必要再实现一个模式。之所以将Abstract Factory保留在模式集合中，是因为第一次解决这个问题时，我认为存在Abstract Factory模式。这也说明了，认为存在一个其实并不存在的模式，未必会有什么副作用。

最后是Abstract Factory模式

13.11 用模式思考：步骤3

完成其余部分

这个设计的细节还需要一些工作，但是，我将遵循Alexander“按背景设计”的要求继续完成该设计。例如，在考虑如何实现SlotFeature类或V1Imp类时，应该记住如何使用相关的模式。在本例中，我注意到在Bridge模式中，与抽象相关的方法独立于实现。这意味着Abstraction类（Feature）及其所有派生类（SlotFeature、HoleFeature，等等）不包含实现信息，实现信息被留在了实现类中。

分配责任

这意味着Feature子类将有getLocation和getLength这样的方法，而实现类将包含一种访问这些必需信息的方法。例如，V1Imp对象需要知道V1系统中Feature对象的ID。因为每个Feature对象有一个唯一ID，这就意味着每个Feature对象都有一个实现对象。V1Imp对象中的方法使用

这个ID向V1Facade查询这个对象的相关信息。

对于V2的实现也存在类似的解决方案。在本例中，V2Imp对象包含一个对OOGFeature的引用。

13.12 与原解决方案的比较

比较各种解决方案

我们来比较一下新的解决方案（如图13-11所示）与原来的解决方案（如图13-12所示）。

比较解决方案的另一种方法

比较两个解决方案的另一种方式是读图。也就是说，图能够形象地展示出继承（is-a关系）和组合（has-a关系）关系。当存在这些关系时，用这些词读图。

原来的解决方案中，由一个Model对象容纳Feature对象。Feature对象可能是沟槽部件、孔部件、方切口部件、不规则形状部件或特殊形状部件。沟槽部件可能是V1沟槽或V2沟槽。V1沟槽使用V1系统，而V2沟槽使用OOGSlot。孔部件可能是V1孔部件或V2孔部件。V1孔部件使用V1系统，而V2孔部件使用OOGHole。真是够枯燥的，不是吗？

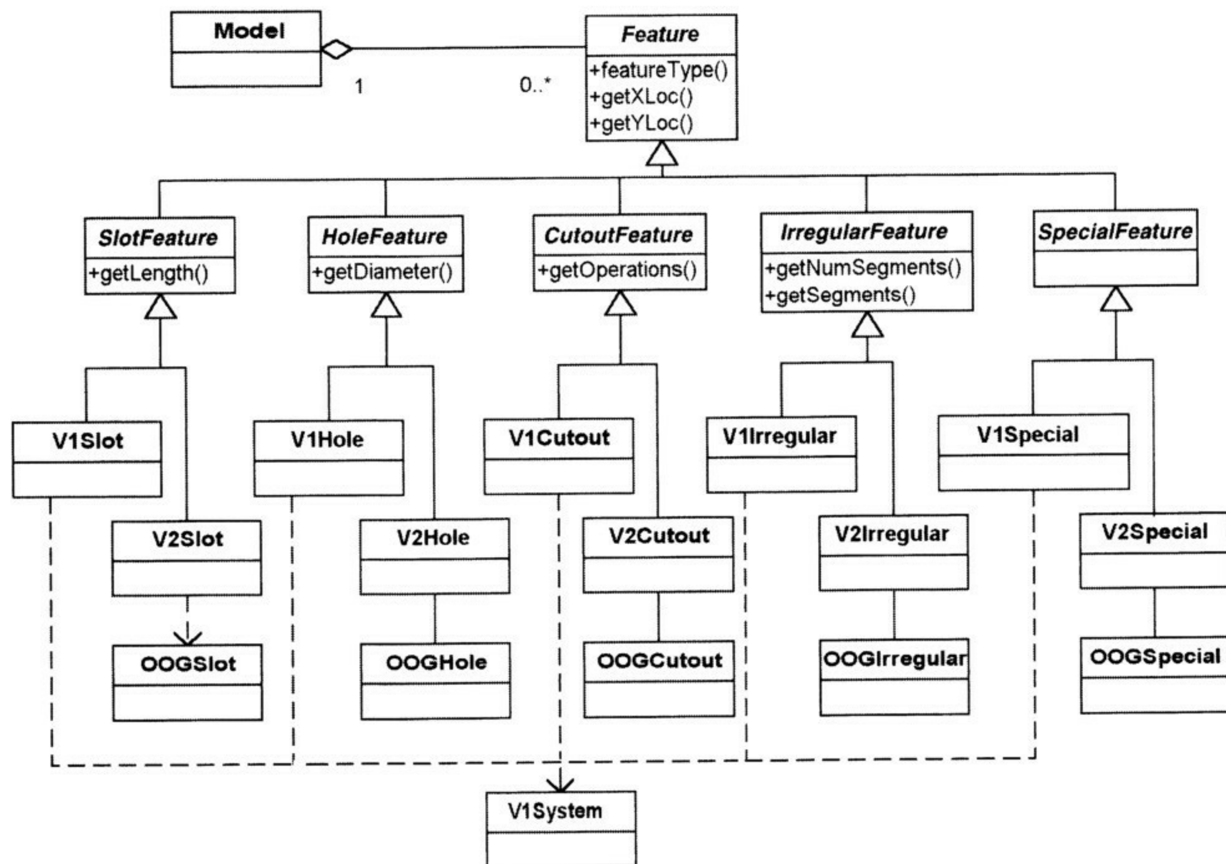


图13-12 第一个解决方案

现在阅读新的解决方案。由一个 Model 对象容纳 Feature 对象。Feature对象可能是沟槽部件、孔部件、方切口部件、不规则形状部件或特殊形状部件。所有的部件都含有一个实现，可能是V1实现或V2实现。V1实现使用V1Facade访问V1系统，而V2实现适配OOGFeature。整个系统比原解决方案的一部分都要简洁多了。

当然，还需要在考虑加入预计的V3系统时，比较两种设计的情况。在原设计中，必须为新的系统派生出新的沟槽、孔等等。而在目前的设计中，只需要创建一个新的适配器（假设V3是面向对象的，这应该比较合理）即可。

13.13 小结

本章内容

本章说明了标准的设计方法经常使我们陷于难以维护的系统之中，通常只见树木不见森林，因为注意力过分集中在系统的细节——类上了。

Christopher Alexander 为我们提供了一种更好的方法。通过在问题域中应用模式，我们可以以另一种方式来看问题：从总体概念开始，然后不断添加特性。运用每个模式都会提供更多的信息。

通过选择能够创建最大的总体概念——系统背景的模式，然后插入第二重要的模式，我开发了一个应用程序架构，这是仅仅观察类所无法看到的。因此，我开始学习按背景设计，而不是将局部相同的部分组合起来的方法。

与第5章中两个木匠需要在鸠尾榫和斜榫之间进行选择一样，背景塑造了设计。在设计决策中，我们经常陷于细节之中，而忘记了系统更大的背景。细节会使我们把注意力放在小而且局部的决策上，它会成为遮蔽更大的总体概念的周围的浮云。模式为我们提供了一种语言，使我们能够超越于细节之上，以实际的方式讨论背景。这样我们更可能看到问题域中的约束因素。模式有助于我们吸取其他开发人员此前了解到的经验教训。因此，它们有助于创建健壮、可维护而且有生气的系统。

复习题

简答题

1.作者所采用的通过模式的软件设计有哪三个步骤？

2.给出背景的定义。

3.给出最高模式的定义。

4.比较两个模式时，区分哪个模式“最高”的两条规则是什么？

5.给出一个模式的一般结构下的定义。它在什么时候有用？

阐述题

1.一个问题总是能用模式定义吗？如果不能，还需要什么？

2.在CAD/CAM问题中，Abstract Factory被排除在“最高”模式之外了。为什么呢？

3.在CAD/CAM问题中，为什么认为Bridge模式高于Adapter模式？

观点与应用题

1.在应用了所有模式之后，还有更多的细节没有处理。这时Alexander的一般原则（从背景开始设计）仍然适用。这种过程会停止吗？什么时候需要继续深入这些细节？这不正是“快速原型”所建议的吗？怎样抵御这种对所有程序员都有吸引力的诱惑呢？是否应该抵御这种诱惑？

2.比较CAD/CAM问题的第一个解决方案（见图13-12）和新的版本（见图13-11）。你更喜欢新设计的哪些地方？

---

[1].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979，p.368。

[2].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979，p.368。

[3].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979，p.368-369。

[4].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979，p.369。

[5].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979，p.370。

[6].这也称为“依赖倒置原则”，是设计模式的一个核心原则。请阅读第14章中的讨论以获得更多信息。

[7].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979 p.372-373。

[8].因为我实际上从来没有问过他，所以准确地说，应该是我所想像的他对此的建议。

[9].关于这一问题的更多信息，可以在Joshua Bloch的杰作Effective Java Programming一书中找到。说得更具体一些，就是书中“创建和销毁对象”那一部分。

[10].V1Model和V2Model之间的差异不会带来太大的困难。因此，我们只一般性地讨论Model类。

第五部分 迈向新的设计方式

概览

本部分内容

虽然模式本身就很有用处，但是从模式中获得的经验还能够用来创造另一种设计应用程序架构的方式，既不同于专注于特殊化的那种方式，也不同于寻找问题域中名词和动词的方式。本部分中，我们将讨论现在如何根据从模式中获得的经验来设计应用程序架构。

章 讨论的主题

14 设计模式的原则与策略

我们对模式已经有了比较多的了解，足以根据来自设计模式的经验创造一种新的设计方式。

15 共性与可变性分析

本章中，我提出一种设计应用程序架构的策略：找出和封装变化。

16 分析矩阵

如何在真实环境中进行共性与可变性分析，从而更加容易地找到和实现各种关系。

17 Decorator模式

说明不应该教条主义地套用模式。过去的实现方式并非模式。推演模式的思考过程对我们更有用处。

第14章 设计模式的原则与策略

14.1 概览

本章内容

前面已经讲述了怎样在局部和全局层次使用设计模式。在局部层次，模式告诉我们如何解决给定背景下的特定问题。在全局层次，模式提供了一张应用程序各组件的关系图。

学习设计模式的方式之一是理解如何在两个层次更有效地运用设计模式。它们将帮助你根据别人的经验开发质量更高、可维护性更好的代码。

学习设计模式的方式之二是，理解其本质机制和它们背后的原则与策略。理解这些知识将提高你的分析和设计能力。即使在还没有发现设计模式的情况下，你也能知道应该怎么做，因为你已经知道了按模式的方式解决这个问题所需的基本概念。实际上，这是在努力理解前人在发现后来成为模式的解决方案时所用的思路。

在本章中，将我们：

叙述开闭原则，这是众多设计模式的基础；

讨论从背景设计原则，这是Alexander模式的目标。在此过程中，本章还将讲述依赖倒置原则和Liskov替换原则[1]，这是设计模式的两个更深层次的基础；

讨论封装变化原则；

讨论抽象类和接口之间的差异；

最后讨论需要对模式保持一种理性的怀疑态度。因为它们都是有益的指导，但并非颠扑不破的“真理”。

14.2 开闭原则

不修改地扩展软件功能

软件显然是需要具有扩展性。然而，修改软件会面临引入问题的风险。这种两难局面使Bertrand Meyer提出了开闭原则（open-closed principle，OCP）[2]。这一原则可以这样理解：模块、方法和类应该对扩展开放，对修改封闭。[3]也就是说，应该将软件设计得不对其修改就能扩展功能。

初听起来这似乎不近情理，但实际上我们已经见过遵守这一原则的例子。例如，在Bridge模式中就很有可能不修改任何已有的类而增加新的实现（也就是扩展该软件）。开闭原则本质上意味着将软件设计成为新功能能够作为单独的模块加入系统，这样就尽量降低了集成的成本。

完全遵守开闭原则几乎是不可能的，但是它可以作为一个目标，指引正确的方向。代码越遵守这一原则，以后适应新（而且可能是无法预测的）需求就越轻松。

14.3 从背景设计原则

模式是 A1exander哲学的缩影

Alexander 教导我们从背景设计，在设计各部分所呈现的细节之前先创建总体概念（big picture）。大多数设计模式都遵循这一方法，而有些设计模式相对其他模式遵循这一方法的程度更深。到现在为止已经讲述的4个模式中，Bridge模式是遵循这一方法的最好例子。

请参考第10章中的 Bridge 模式图（见图 10-13）。在决定如何设计Implementation类的接口时，需要考虑其背景：从Abstraction派生的类如何将使用这些Implementation类。

例如，如果所编写的系统需要在不同类型的硬件上绘制各种几何形

状，因此需要不同的实现，我就会使用Bridge模式。Bridge模式建议：几何形状将通过一个公共接口使用该实现部分（也就是要编写的绘制程序）。像Alexander教导我们的那样从背景设计，也就意味着我应该首先理解几何形状的需求，即要绘制什么？这些形状将决定实现所需的行为。例如，实现（绘制程序）可能必须绘制直线、圆等。换言之，即使具体的抽象（不同的几何形状）依赖于具体的实现（绘制程序），仍然是形状定义绘制程序。应该从面向服务开始，问一问："我们要提供哪些种类的服务？"因为是从 Abstraction类（及其派生类）定义，所以必须在抽象层次定义服务。在为实现开发接口时，我要考虑具体抽象中所说明的概念的需求，而不仅仅是所面对的特殊情况。

依赖倒置原则：依赖抽象！

这就是所谓"依赖倒置原则"（dependency inversion principle，DIP）。

高层模块不应该依赖于低层模块。高层模块和低层模块都应该依赖抽象。

抽象不应该依赖于细节。细节应该依赖于抽象。[4]

Christopher Alexander 将此称为"复杂化"——一种从最简单（概念性）的层次开始，然后逐渐添加细节和特征，随着逐步深化，设计也渐趋复杂的过程。复杂化和依赖倒置是使用设计模式的中心基础原则。

这一原则隐含着使用对象和被使用对象之间只能在概念层次存在耦合，而非实现层次，这与《设计模式》一书中所建议的应该"按接口设计"可以说是英雄所见略同。在这个层次考虑新的服务和新的客户对象的关系，比面对如何提供服务的细节知识时考虑要容易得多。

从背景设计的优点

我将花一些时间找出一般情况和可能遇到的各种变化。这需要考虑类所处的背景，并使用一种称为共性和可变性分析的技术。该技术将在第15章中详细讲述。这能够帮助我根据进一步一般化（generalization）

所需的成本，决定实现的一般化程度。这往往能比其他思考方式带来更具一般化的实现，所增加的成本很小。

例如，在考虑绘制几何形状的需求时，我可能很容易就找出直线和圆是需求。如果我自问：“有了直线和圆之后，什么形状还无法支持？”我可能会注意到还不能实现椭圆。现在可以这样选择：

除了直线和圆，再实现一种绘制椭圆的方法；

认识到椭圆是圆的一种一般情况，改而实现椭圆；

如果成本大于预期的收益，就放弃实现椭圆。

从背景设计的另一面

设计在所处的背景中外部行为相同的多个类时，这一点非常重要。这能够将客户类与这些实现解耦，还可以如设计模式所建议的提升类型的封装性。Barbara Liskov于1988年提出了一个原则[5]，复述如下：

一个从基类派生的类应该支持基类的所有行为。

只要可能，我喜欢将这一原则扩展为：让使用对象甚至无法知道是否存在派生类。也就是说，对于给定的基类（或者接口）的引用，使用对象无法知道其是否存在派生类（或者实现类）。因此所有这样的派生类（或者实现类）都是彼此可以互换的，从而对类型进行了很好地封装。实践中这意味着子类型不应该在基类型的公开接口中添加新的公开方法。这还意味着基类型必须是所建模的概念的完整规格说明。

前面的例子说明了另一重要的设计概念：

发现可能性不意味着必须按其行事

只是可能存在并不意味着必须要实现。我的设计模式经验表明，设计模式赋予了我对问题领域的洞察力，但是，并非总要（甚至往往不需要）真的根据这种认识去做，为还没有出现的情况编写代码。当然，这些模式有助于从背景设计，它们使我能够预测到可能的变化，因为已经将系统分为多个强大而且易用的抽象，因此更容易适应变化。设计模式有助于看到什么地方可能出现变化，而不是会出现哪个具体的变化。我

用来封装当前变化的具有明确定义的接口经常也会限制新的需求带来的影响。[6]

Abstract Factory 模式的背景

Abstract Factory模式是另一个从背景设计的好例子。开始可能只知道可以用某种工厂对象协调一系列（或一组）对象的实例化，但是，现在明白它还有许多不同的实现方式。

实现方式	说　明
使用派生类	经典的 Abstract Factory 模式实现建议为需要的每组对象实现一个派生类。这有些笨拙，但也有其优点，可以不改变任何已有类而增加新类
使用含有 switch 语句的一个对象	如果希望按需要修改 Abstract Factory 类，可以只让一个对象包含所有这些规则。尽管这并不遵循开闭原则，但是它在一个地方包含了所有的规则，不难维护
使用含有 switch 语句的配置文件	这比前一种方式更加灵活，但有时仍然需要修改代码
使用含有动态类加载的配置文件	动态类加载是一种根据字符串中的对象名实例化对象的方法。这种实现灵活性很强，可以不修改任何代码增加新的类和新的组合

如何决定？根据背景

对所有这些选择，应该怎样决定用哪种方案实现Abstract Factory呢？根据模式出现的背景来决定。根据以下因素的不同，这4种方式各自都其优于其他方式之处。

未来变化的可能性。

不修改当前系统的重要性。

谁控制将要创建的对象集合（我们还是另一个开发组）。

使用何种编程语言（比如：C++, C#, Java）。

是否有数据库文件或配置文件。

这个列表当然并不完整，前面的实现方式列表也是如此。但是，有一点应该很清楚了：在不懂得怎样使用Abstract Factory模式（也就是说不了解其背景）之前要想决定其怎样实现，显然是徒劳无功。

怎样进行设计决策

> 尝试在不同实现方式进行选择时，许多开发人员会这样问：“这些实现方式中哪个更好？”这并不明智。问题在于，往往没有哪个实现方式天生优于另一个。应该这样问，对于每种实现方式，“什么情况下它优于其他方式？”然后再问：“哪种情况与我的问题领域最相似？”这种反复思考的工作量并不大。这种思考方法能够使我更明了问题领域中的变化和可伸缩性问题，而且不会在得到第一个、可能不完整的答案之后就停下来。

Adapter模式的背景

Adapter模式也说明了从背景设计原则，因为它几乎总是出现在某个背景中。根据定义，Adapter类的作用是将一个原有接口转换到另一个接口。有一个问题其答案显而易见：“我怎么知道应该将原有接口转换成什么样呢？”在背景（也就是要适配的类或者抽象）显现之前，通常都不会知道。

我已经说明了Adapter类可以用来转换一个类以适应一个已有模式。这正是CAD/CAM问题中的情况，其中我有一个已有的实现，需要将它转换为Bridge模式驱动的实现。

Facade模式的背景

Facade模式与Adapter模式就背景而言非常相似。一般Facade模式是在其他模式或类的背景中定义的，也就是说，必须等到知道谁想用Facade模式设计自己的接口。事实上，Facade的接口经常是随着使用它的系统部分不断开发出来而逐步开发出来的，开发期间每个新的系统部分都为Facade建立了更加丰富的背景。

一个警告

在刚刚开始使用模式时，我总是认为自己可以发现哪些模式在为其他模式创造背景。在Alexander的Pattern Language一书中，他对于建筑模

式就能够这样。因为很多人都在谈论软件的模式语言，我想：“我为什么不行呢？”似乎很显然，Adapter模式和Facade模式总是在其他模式所创造的背景中定义的。对吗？

错。

同时负责教授别人的软件开发人员有一个好处，就是有机会比纯粹的开发人员接触更多项目。在刚开始教授设计模式时，我认为Adapter模式和 Facade 模式总是在其他模式定义背景的非创建型模式之后出现。事实上确实经常如此。但是，有些系统需要构造一个特定的接口。这时，Facade模式或Adapter模式就可能成为最高模式。

14.4 封装变化原则

我的设计的一点备注

有些读者可能注意到我的所有设计中都有一个相似之处：继承层次中类很少超过两层。那些层次更多的设计往往都是因为有的设计模式的结构要求有两层作为派生类的基础。（第17章中讨论的Decorator模式是一个使用了三层的例子）。如果我出现了多层继承的，几乎总是为了要消除某种冗余（当然，对于Decorator模式，Decorator抽象类的存在是为了在一个地方存放装饰与被装饰物的关系。）。

之所以如此，是因为我的设计目标之一就是不让一个类封装两个要变化的事物，除非这些变化明确地耦合在一起（比如，一个数据库的多个实现方法）。这样会降低内聚性，变化之间的耦合也无法松散。迄今为止所讲述的模式已经说明了有效地封装变化的各种不同方法。

用 Bridge 模式中封装变化

Bridge模式是封装变化的一个绝佳范例。Bridge模式中的实现除了能够通过一个公共接口访问之外完全不同。新的实现部分可以通过在此接口中实现相应功能系统加入。

用 Abstract Factory模式中封装变化

Abstract Factory 模式封装的变化是哪些系列或哪些组对象可以实例化。这个模式的实现有很多种方式。需要指出的是，即使一开始选择了某种实现方式，然后发现另一种方式更好，可以在不影响系统其他部分的情况下改变其实现（因为工厂对象的接口不变，只是其实现方式改变）。因此，Abstract Factory模式概念本身（按一个接口实现）隐藏了如何创建对象上的所有变化。

用Adapter模式帮助封装变化

Adapter 模式可以用来给截然不同的对象定义一个公共的接口。现在我经常要用到它，因为其他许多模式都要求按接口设计。

在Facade模式中封装变化

Facade 模式一般不封装变化，但是，我见过很多情况下要用 Facade 模式处理一个特定的子系统。然后，当另一个子系统出现时，新子系统的Facade 也要按相同的接口创建。这个新类结合了 Facade 模式和Adapter模式，因为主要动机就是简化，但现在还有一个附加条件：要与前面已经使用的接口保持一致，从而保证客户对象都无需修改。这样使用 Facade模式就隐藏了所用子系统上的变化。

不仅封装变化

但是，模式并非只能封装变化。它们还有助于找到对象之间的关系，帮助我们思考问题域中的关键概念。还是来看Bridge模式，注意到了吗，这个模式不仅定义和封装了抽象和实现中的变化，而且还定义了两组变化之间的关系。

14.5 抽象类与接口

抽象类和接口的表面区别

抽象类和接口之间的一个区别，就是抽象类允许有公共的状态和行

为。也就是说，如果所有派生类都有一些公共的状态或者行为，就可以放在抽象类中。对于Java和C#这样的语言而言，进行这种区分尤其重要，因为这些语言只允许从一个类继承。换言之，在不需要的时候不要使用抽象类，因为只有从一个类派生的机会。

抽象类与接口在设计层次上的比较

对于这两种获得多态性的机制之间的区别，还有另外一种思考方式对设计人员更有帮助。它们的区别在于本章前面提到的各原则的背景上。抽象类可以看成是一种聚集相关实体的方式。其关注点是如何设计这些具体的实体（派生类），从而可以以同样的方式使用它们。也就是说，如何保存这些实现并封装起来。此处的关注点仍然遵循着依赖倒置原则：考虑服务对象（实现），看如何抽象它们，使用对象才不会与任何特定于实现的细节相耦合。

而为了这样使用接口，在设计时应该问的是："这些东西如果要以相同的方式使用，必须都有什么样的公共接口？"

但是还必须考虑另一方面，接口的关注点是要使用这些派生/实现的对象。也就是说，服务对象应该有什么样的接口，才能最好地服务于背景/控制对象？在前面电子商务的例子中，问题就变成了："计税对象应该有什么样的接口，才能最好地服务于SalesOrder对象？"这在依赖倒置原则基础上更进了一步。

从客户对象开始，可以将"被使用的"对象分化为更小、内聚性更强的部分。换言之，如果使用对象要求许多不同类型的对象（即多个抽象），可以为每种类型设置一个接口。而结果将是比其他方式更简洁的接口。[7]

各有所长

不能仅仅因为接口看上去更符合依赖倒置原则，或者接口更简洁，就认为接口优于抽象类。

按照这种思路，你很可能在开发了一个接口之后，却发现应该使用

抽象类，因为已经定义的对象有一些公共的状态/行为，如果将它们放在实现类中就会出现冗余。

也可能设计一个接口，然后用一个抽象类实现该接口。

抽象类能够确定默认行为，从而使实现类更加简单，维护起来也更容易。

这样，具有公共状态或行为的对象从这个抽象类派生，而不直接共享这一状态或者行为的对象（或者由于其他原因必须从另一个类派生的对象）实现接口。

14.6 理性怀疑原则

模式是有用的指南但也是危险的工具

基于模式的分析方法已经用于许多学科。在作者本人从事的人类学和知识工程学领域，分析人员已经获得了许多可以用模式表达的教益。模式本身是非常有用的，但是应该将它们用作一种思考问题的辅助手段，而不是解决问题的处方。这一点无论怎么强调都不过分。

那些概念层次（meta-level）的模式和模型都不是真理，它们只是真理的抽象。它们是已往经验和教训的结晶，应用于真实世界必须具体问题具体分析。在使用模式时，有如下常见错误。

错　误	产生条件
浮于表面	仅仅对低层情况有了一些肤浅的理解，就草草选择一个模式
偏见	对模式过于偏信。根据已经选定的模式/模型来解释所有数据，不愿意对自己的偏见有任何质疑
错选	不理解模式适用的背景和条件（对各模式的分类关系理解不全），选择了错误的模式
误判	不熟悉各种模式，因为无知而导致误判
削足适履	忽略了实际的、具体实例行为中的例外情况，因为它们似乎不符合模式中所表达的理论。很可能会使所建模出来的对象过于僵硬，不符合实际情况

请记住模式都是发现而不是发明出来的，这一点也很重要。“适合”某个问题的模式就在问题之中，而不是强加在问题之上。与此类似，模式实现的具体方式应该由问题的本质、约束条件和需求等等决

定，而不是根据你在某本模式书中碰巧看到的某个实现。

14.7 小结

本章内容

本章中讨论了模式如何阐释两个强大设计策略：

从背景设计

在类中封装变化

这些策略使我们可以在看清了各种可能的决策之后再做定夺。观察所处的背景，能够提高设计质量。

通过封装变化，可以适应许多未来的变化——如果不努力使设计更加通用，这些变化出现时将无法适应。对于那些无法获得需要的所有资源的项目（世上的所有项目莫不如此）来说，这是至关重要的。通过适当地封装变化，可以只实现那些需要的特性，而无需牺牲未来的功能。试图确定并适应所有可能的变化通常并不能获得更好的系统，这样往往会一事无成。这就是所谓的分析瘫痪（paralysis by analysis）。

原　　则	描　　述
开闭原则	模块、方法和类应该对扩展开放，对修改封闭。换言之，软件应该设计成不加修改原有代码就能扩展功能
依赖倒置原则	其背后的理念是应该在设计细节之前先创建总体概念。高层模块不应该依赖低层模块。相反，它们都应该依赖于抽象
理性怀疑原则	小心过分依赖模式。概念层次的模式和模型都是真理的抽象。它们是已往经验和教训的结晶。使用它们来帮助我们思考摆在面前的问题

复习题

简答题

1.在选择如何实现一个设计时，应该提什么问题？

2.使用设计模式时的五种可能的错误是什么？能够对它们阐述一下吗？

阐述题

1.“开闭原则”规定：“模块、方法和类应该对扩展开放，对修改封闭。”这是什么意思？

2.Bridge模式从哪一方面说明了开闭原则？

观点与应用题

1.虽然设计模式可以使我们深入地观察各种情况，但是并不是非要编写处理所有可能性的代码。怎样决定哪些可能性要现在处理，哪些应该为未来做准备？

2.根据你目前的工作，举一个错误使用设计模式的具体例子。

第15章 共性与可变性分析

15.1 概览

本章内容

本章说明如何使用共性和可变性分析（commonality and variability analysis，CVA）开发高层的应用程序设计。虽然设计模式并不能用于所有设计之中，但是它们提供的教益是普适的。这些教益中最重要的一条就是可以使用 CVA 找到系统中的变化。然后就可以按照设计模式的其他教益（按接口编程、使用聚集封装变化）获得灵活和易于测试的设计。

15.2 共性和可变性分析与应用程序设计

隔离变化是设计模式的理念之一

有经验的开发人员都知道，需要在现有系统中增加新功能的时候，主要的成本往往不是在编写新的代码，而在如何将它集成到原有系统中。其原因在于，大多数原有系统中的各个组成部分是紧密耦合的，必须消除或者尽量限制这种耦合。导致这种紧密耦合的原因就是开发人员经常在弄清楚实体本身之前就考虑实体之间的关系。从我们培训的各种水平开发人员的经验来看，可以得出这样的结论：经验丰富的开发人员比缺乏经验的开发人员更容易犯这种错误。开发人员需要一种方式，首先弄清有些什么东西，然后再尝试找到它们之间的关系。

我建议，设计程序时应该按照这样的方式进行：首先，使用 CVA 找到问题域中存在的各种概念（共性）和具体的实现（可变性）。这时

我们最感兴趣的是找到其中的概念，但是这一过程中也会发现许多可变性。问题域中任何没有包含在这些概念中实体（比如可能有一些属于“某种”对象的对象）也应该找出来。然后，在所需功能的概念都找到之后，继续为封装这些概念的抽象制定接口。接着考虑你将如何使用从该抽象派生的具体实现，根据这一点派生接口。

这种方式基本上就是遵循了Alexander的背景设计方法，这种方法蕴涵在上一章提到的依赖倒置原则里。通过定义这些接口，还确定了哪个对象使用哪些对象——从而完成了设计的规格说明。

让我们通过CAD/CAM问题来实际体验一下。

15.3 用CVA解决CAD/CAM问题

寻找概念，进而寻找抽象类

用 CVA 分析问题领域时，需要分析存在哪些概念，然后尽可能紧凑地组织这些部分。还记得CAD/CAM系统吗？其中有：

不同的CAD/CAM系统，即V1和V2。在这里，实际上也就是只读而且专有的数据库，它们为专家系统提供其工作所需的数控集；

不同类型的部件，即沟槽、孔、方切口、特殊形状和不规则形状；

不同类型的模型，即基于V1的和基于V2的。

这里所谓的不同 CAD/CAM 系统，实际上意味着存在概念“CAD/CAM系统”及其变化V1和V2。通过CVA能够得到表15-1中列出的共性及其相应的变化。

表15-1 共性和可变性分析表

共　　性	变　　化
CAD/CAM 系统	V1
	V2
部件	沟槽
	孔
	方切口
	特殊
	不规则
模型	基于 V1 的
	基于 V2 的

另一种方法是选择问题域中的任意两个东西，然后问以下问题。

其中一个是另一个的变化吗？

它们都是其他东西的变化吗？

例如，我注意到有部件和沟槽。沟槽是一种部件。我猜测“部件”应该是一种共性，而“沟槽”是它的一种变化。也可能，我看到有沟槽和孔。在这个问题域中，它们似乎互相都不是彼此的变化，它们似乎都是“部件”的变化。我也可能比较V1 CAD/CAM 系统和沟槽，它们之间似乎没有任何共同之处。

每个共性一个问题

当然，事情并非总是这么简单。有些概念可能会被遮住。例如，假设我们考虑的问题域有V1Slot、V1Hole、V2Slot和V2Hole等。共性似乎是“CAD/CAM部件”。但是这个共性有两个概念：“CAD/CAM版本”和“部件”。CVA告诉我们共性有一个原则：每个共性一个问题。否则设计中就不能具有比较强的内聚。既然认识到存在两个共性，CAD/CAM和部件，就应该问这些共性都有哪些变化？这样就得到了表15-1 中列出的变化。这正是CVA的价值之一：它能够得到紧凑的设计。

概念的表示

图15-1是图8-5的复制。

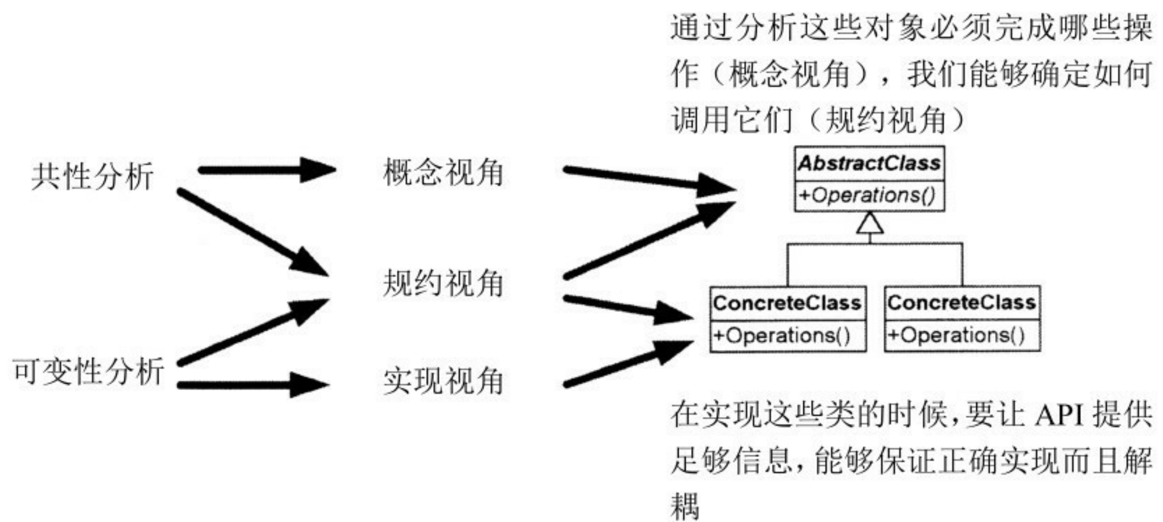


图15-1 共性和可变性分析、三种视角和抽象类之间的关系

根据图15-1中所提出的原则，表15-1中的信息可以转换为三个不同的类层次。如图15-2所示。

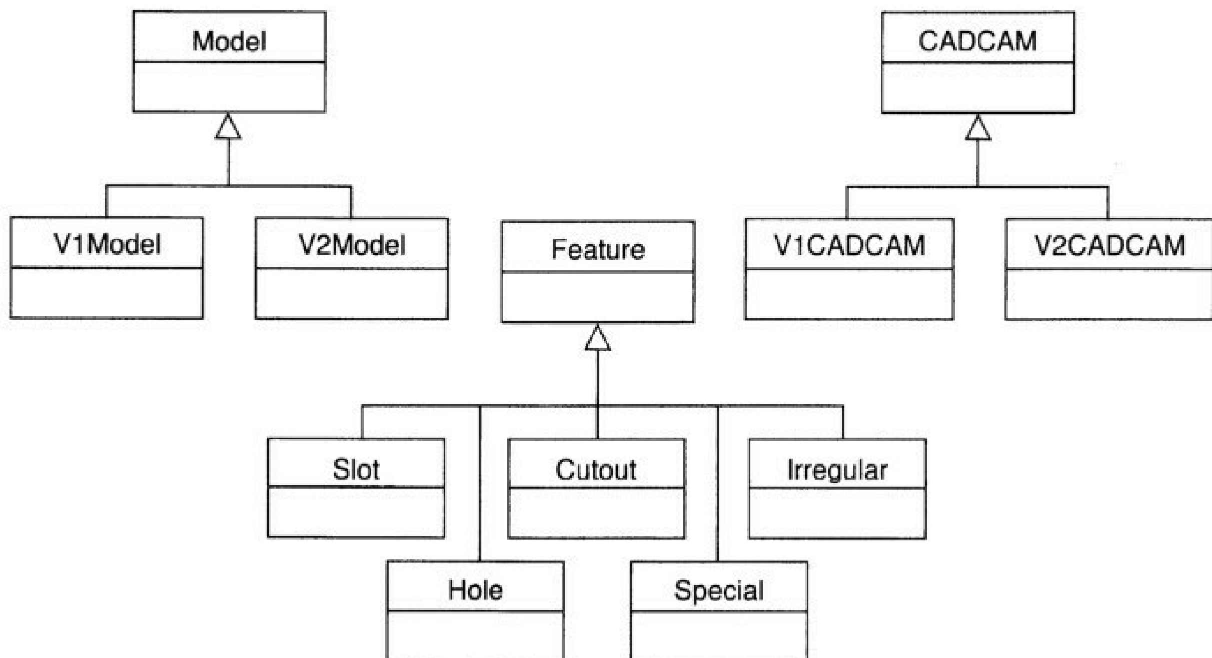


图15-2 将CVA表格转换成类

概念的关系

下一步应该确定概念之间的关系。模型包含部件，而部件是从

CAD/CAM系统提取出来的。当时设计这个系统时，我认为如果构建单独的部件，包含CAD/CAM系统中所有与之相关的信息，应该更简单。也就是说，CAD/CAM系统对于部件而言就是数据库，它保存有关部件的信息。部件应该有相应的方法，以提供这些信息，但是部件需要从CAD/CAM提取该信息。模型还必须与CAD/CAM系统有关系。基于这些分析，下一个层次的细节就显现出来，如图15-3所示。

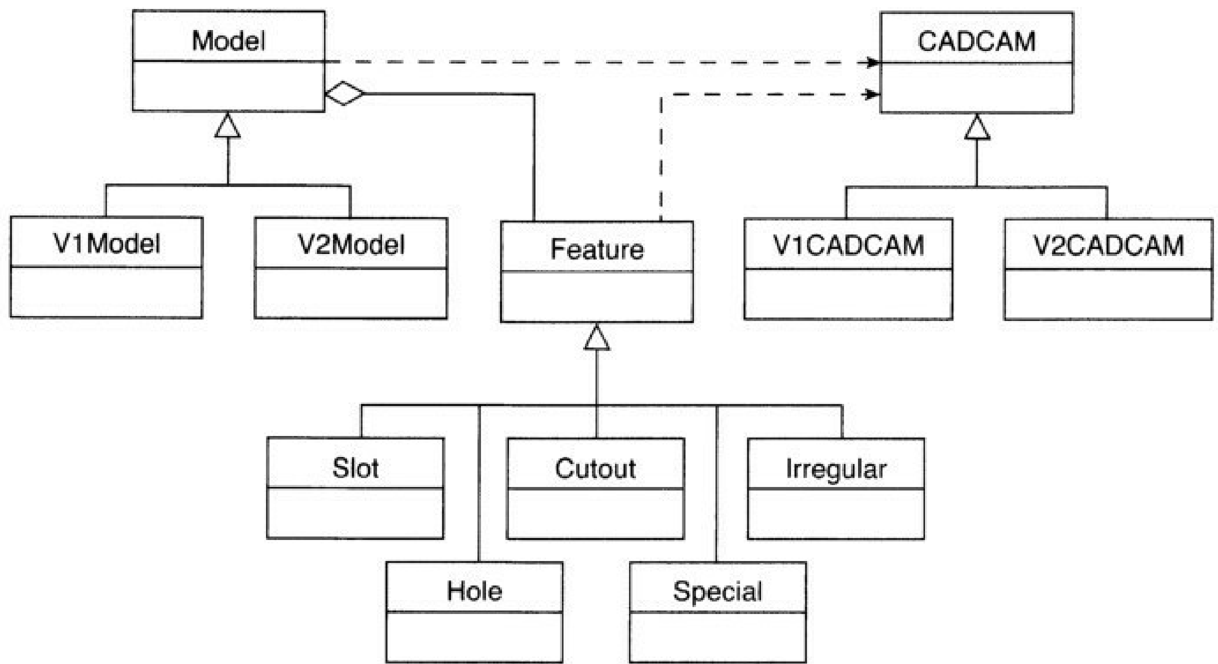


图15-3 表示了类之间关系的类图

这里面临着一个设计决策。应该是用不同类型的模型，还是让一种类型的模型通过 CADCAM接口使用不同的 CAD/CAM 系统?也就是说，如果将V1Model和V2Model中特定于CAD/CAM系统的方法移到CADCAM类层次，可能可以避免拥有不同类型的Model。这种方法看起来似乎更好，因为Model可以使用CAD/CAM系统实现，但问题是Model中的概念并非CAD/CAM系统本身所有的。这种方法如图15-4所示。不要在两种方法之间的区别上钻牛角尖。实际上只要确保没有冗余，二者就代码质量而言并无大的区别。我个人更喜欢图 15-4 中所示的方案，

因为它的类更加紧凑。所以，接下来我们就以这一方案为例讲述。

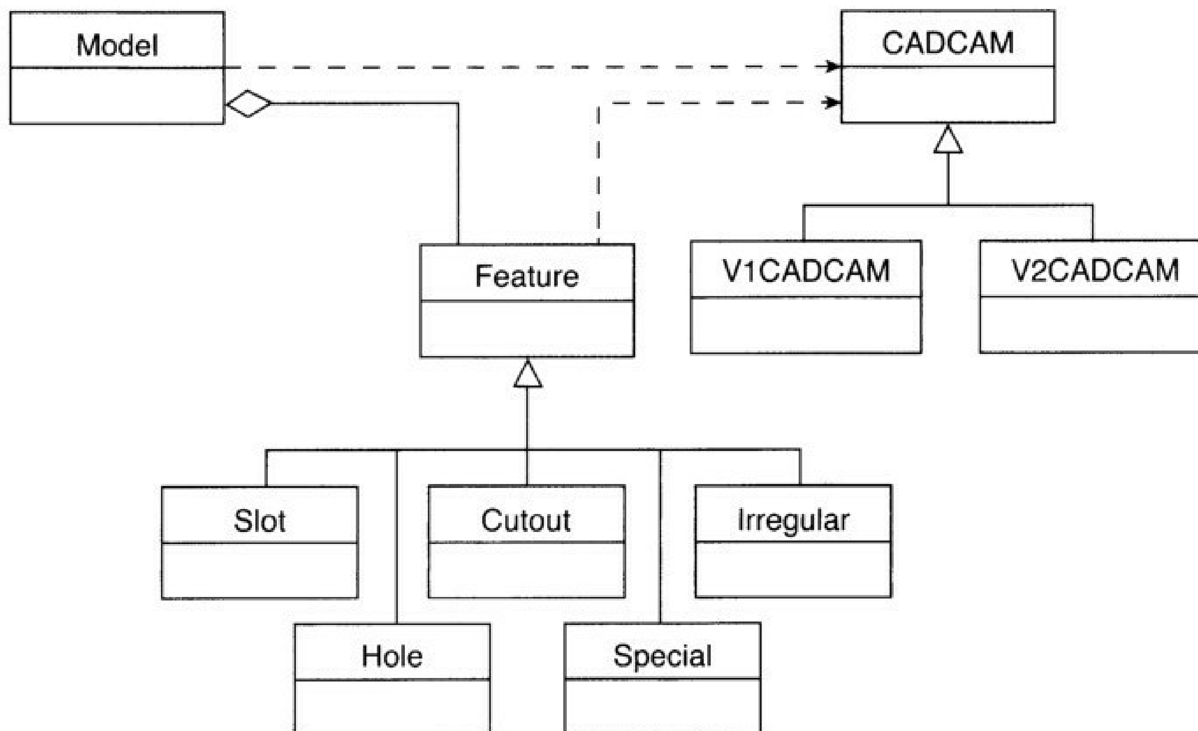


图15-4 通过使用CADCAM 处理Modle中的变化

扩展设计

接下来还需要扩展设计，将CADCAM类和实际的V1和V2实现联系起来。回想一下我们从Facade和Adapter模式所学的知识，显然V1CADCAM应该就是 V1 系统的 Facade，而 V2CADCAM 应该包装（适配）V2 系统（OOG_Part），如 图15-5所示。

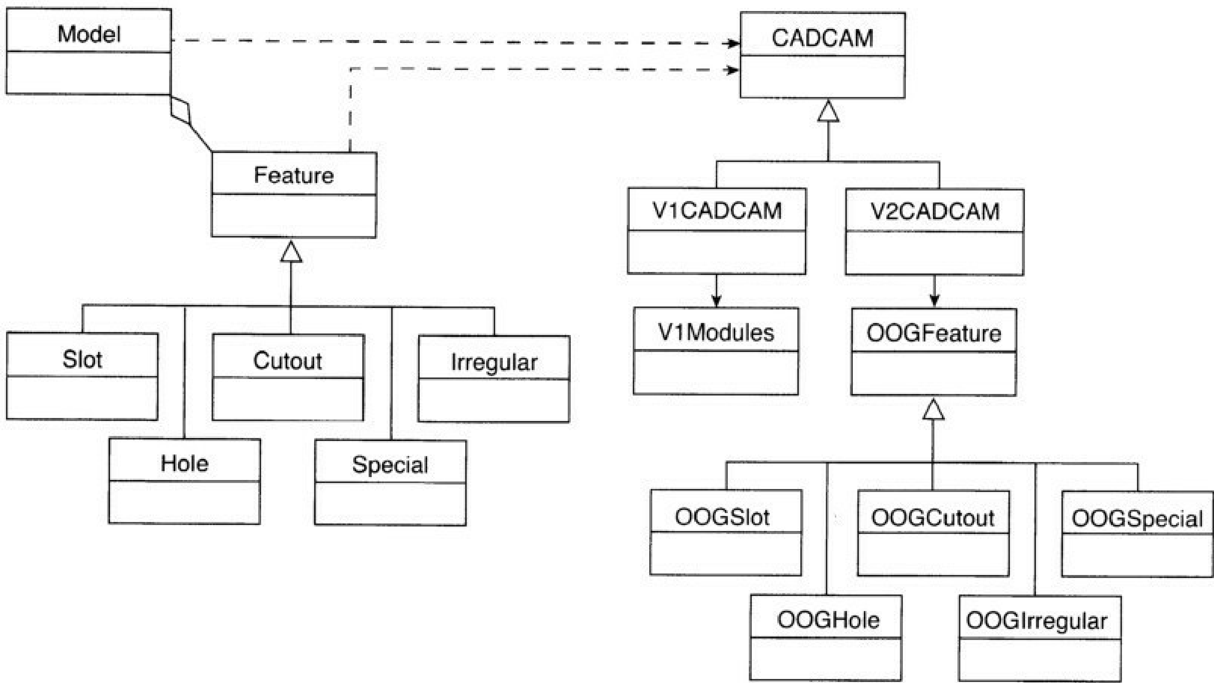


图15-5 完整的设计

将这个方案与直接使用设计模式得到的方案（如图 15-6 所示）比较一下。

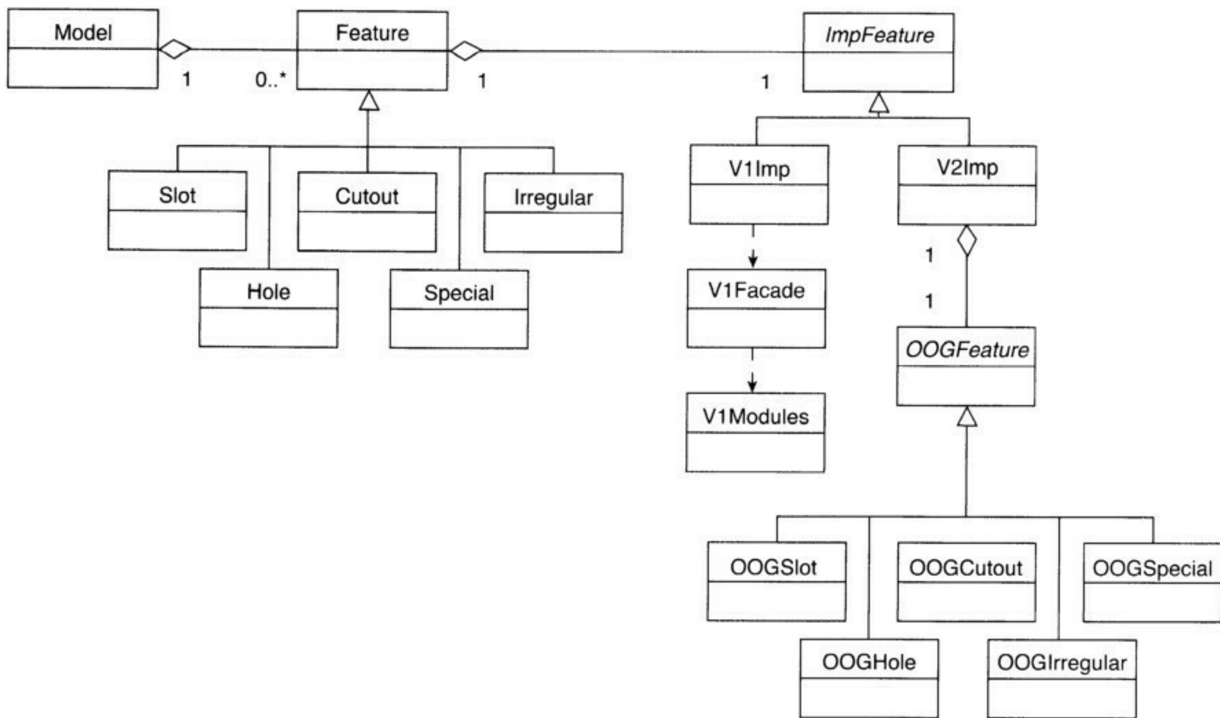


图15-6 使用设计模式的方案

两者看上去惊人地相似。噢，还不尽然。使用设计模式获得的方案是以背景方式使用模式得出的，我每次应用一个模式，直到解决方案完全显露出来。这与CVA方式是非常类似的。

1.先寻找共性。

2.从这些共性创建抽象。

3.从共性的变化寻找派生。

4.看共性之间的关系如何。

这是另一种形式的从背景设计。类的接口是在其他抽象如何使用这些类的背景中定义的。两者方法中的类定义是类似的，因为 CVA 只是另外一种寻找变化、并用高内聚、松耦合的类封装（设计模式的基础原则）的方式。所以相同的原则和方法会得到非常类似的解决方案。

实际上，这两种方法是相辅相成的。CVA强调尽早关注抽象，这样更可能找到最有用的抽象。设计模式关注于这些抽象之间的关系，但是对找出最重要的抽象帮助不大。

从另一方面来说，设计模式可以使我们利用来自过去成功设计的真知灼见，而CVA做不到这一点。例如，我知道应该使Facade无状态，因为它很可能比较大，创建多个实例会影响性能。从 Facade 中将部件所需的状态提取出来，放入Adapter中，可以将Facade实现成一个Singleton。 CVA可无法使我得出这样的结论。

15.4 小结

本章内容

我们阐述了怎样使用 CVA 创建应用程序的高层设计。通过先定义共性，消除了特殊情况之间的耦合。因为设计模式的本质实际上就是隔离变化和共性，这与共性和可变性分析（按我所说的方式使用）异曲同工，运用共性与可变性分析和遵循设计模式将得到相似的解决方案。但

是，CVA方法的优势在于，它是普遍适用的。而我们只能在知道存在模式的时候运用模式进行设计。我的经验表明这并不会经常发生。

复习题

简答题

找出共性和可变性的两种方法是什么？

阐述题

1.CVA要求我们每个共性一个问题。为什么这一点非常重要？

2.CVA和设计模式是怎样相辅相成的？

观点与应用题

1.有经验的开发人员甚至比经验并不多的同行更经常地在弄清楚实体本身之前，过早关注实体之间的关系。你的经验是这样的吗？举出一个例子支持或者反驳这一说法。

2.说明从CVA开始的设计方法与Alexander方法的关系。

第16章 分析矩阵

16.1 概览

本章内容

本章将继续前面第9章开始的电子商务案例研究的讨论。

现在我们已经讨论了许多模式，应该回过头来，考察软件开发中最大问题之一：处理问题域中的变化。设计模式可以帮助分析师成功地找到变化并很好地将其组织起来。

在本章中，我们将：

思考现实世界中的变化问题；

考察电子商务案例研究中代表着问题主要变化的部分。在解决这一问题的过程中，形成一个分析矩阵，它是决策表的一种简单变形，我发现它对于理解和协调概念中的变化很有帮助。分析矩阵和Christopher Alexander、Jim Coplien的概念有异曲同工之妙；

描述如何在实际中使用分析矩阵。

16.2 现实世界：充满变化

现实世界中的更多变化

现实世界中问题往往不会秩序井然或者循规蹈矩地出现。除了那些最简单的问题，似乎总会有一些没有什么规律的异常和变化。它们是一些异常的不良特性（gotcha），会破坏我们精心设计的模型。

例如，病人去医院一般会先去挂号处。但如果情况紧急，可能危险生命时，病人可以直接进入急救室，不必先挂号。这些就是现实世界中

的变化，我们的系统必须处理的各种不同的特殊情况。

这也正是使我们这些分析师头疼的东西。模式可以帮助我们更有效地应对变化吗？

我使用了一种方法，可以明确系统中的变化，然后使用这种分析找到应该在设计中使用什么模式。我的方法步骤如下所示。

1.找到某种特定情况中最重要的特性，并用矩阵将它们组织起来。用特性所表示的概念为每个特性标记。

2.继续处理其他情况，按需要扩展这个矩阵。处理每一情况时应该独立于其他情况。

3.用新的概念扩展该分析矩阵。

4.用行发现规则。

5.用列发现特定情况。

6.从这种分析中确定模式。

7.得到高层设计。

16.3 国际电子商务系统案例研究：应对变化

电子商务系统：一个说明变化的案例

我们现在再来讨论第9章中介绍的案例研究，一个美国某国际电子商务公司的订单处理系统。假设我们的电子商务系统必须能够处理来自不同的国家（地区）的销售订单。本章我们将考虑应对这种变化。

最开始，需求很简单：只处理美国和加拿大的订单。系统必须处理的特性清单如下，顺序不分先后。

要为加拿大和美国构建一个销售订单系统。

根据所在国家计算运费。

运费还应该以所在国家（地区）的货币支付。

在美国，税额应按当地计算。

使用美国邮政规则验证地址。

在加拿大，使用联邦快递发货，同时缴纳联邦政府销售税（Government Sales Tax，GST）和地方销售税（Provincial Sales Tax，PST）。

首先要做的，是将这些需求分成两种情况：

顾客在美国；

顾客在加拿大。

将这些都记在表16-1中。

……说明分析矩阵技术

这个问题中存在的变化并不太复杂。光凭观察，应对方法也显而易见。这是一个简单的问题，的确如此。但它说明了一种我已经多次使用过的应对变化的技术。这个技术很简单，但是能够很好地扩展，用于很多真实问题。我将这种技术称为分析矩阵（analysis matrix）。

表16-1 因顾客所在地不同的两种情况

情　况	过　程
美国	根据 UPS 费用计算运费 使用美国邮政规则验证地址 按当地计算销售额和/或服务的税额 用美元处理金额
加拿大	用加拿大元处理金额 使用加拿大邮政规则验证地址 通过联邦快递发货到加拿大 按加拿大省的税收规则计算销售额和/或服务的税额（使用 GST 和 PST）

分析矩阵的起源

虽然这里所讲述的例子非常简单，但我是为了解决一个极大的问题而发明分析矩阵的。那个问题中，有成百上千种情况，50　多种变化。我发现自己当时甚至无法与项目的分析师交谈，因为信息实在是太多了。认识到没有老路可走之后，我清楚自己需要提出一种组织海量数据的新方式。

为此需要，我创造了这里所述的分析矩阵。一开始，我只是尝试用它组织数据。但是，本章后面你可以看到，当问题域围绕变化组织之后，它使我们更容易看到如何使用设计模式创建应用程序的高层设计。也就是说，分析矩阵不仅帮助我们理解问题域，而且有助于实现它。

1.找到某种特定情况中最重要的特性，并用矩阵将它们组织起来

本步骤的目的是寻找会发生变化的概念，寻找共性之处，发现漏掉的需求。这些概念来自每种情况的具体需求。设计和实现的问题将在以后的步骤中讨论。

让我们从观察一种情况开始。

首先观察必须实现的每个功能，并标记它所表示的概念。每个功能点将分行写出，这一行的左边写上它所表示的概念。

从表16-2开始，我将一步一步地给出整个过程。

表16-2 填写分析矩阵：第一个概念

	美国销售
计算运费	按照 UPS 费率

现在，继续下一信息："使用美国邮政规则验证地址"，为此信息增加一行，如表16-3所示。

然后处理第一种情况的所有概念，如表16-4所示。

表16-3 填写分析矩阵：第二个概念

	美国销售
计算运费	按照 UPS 费率
验证地址	使用美国邮政规则

表16-4 填写分析矩阵：完成第一种情况——美国销售

	美国销售
计算运费	按照 UPS 费率
验证地址	使用美国邮政规则
计算税额	使用美国和当地的税收规则
金额	美元

2.继续处理其他情况，按需要扩展这个矩阵

现在转到下一种情况和其他情况，每种情况一列，根据已有的所有信息完成每个单元格。按需求所提供的顺序进行。无需对各种情况进行比较，但是需要将新情况与正在处理的概念进行比较。记住一点很重要，需要考察的是这些新情况对最左一列中已经找到的概念如何进行处理。

还应该记住，构建矩阵时首先应该考察从 CVA 得到的关系，这时并不需要寻找其他关系。我们来填表，首先添加了针对加拿大情况的一列，如表16-5所示。

表16-5 下一种情况的分析矩阵——加拿大销售

	美国销售	加拿大销售
计算运费	按照 UPS 费率	
验证地址	使用美国邮政规则	
计算税额	使用美国和当地税收规则	
金额	美元	

第一条需求是：用加拿大元处理金额。应该放在哪一行呢？我认识到计算运费时需要使用加拿大元，但是加拿大元并不是“计算运费”的一种，因此不应该放在这一行。加拿大元和“验证地址”之间看不出任何关系，所以再看下一行。“计算税额”与“计算运费”的情况相同，有关系，但不是CVA关系。呵呵，最后一行是“金额”，加拿大元正是共性“金额”的一种变化，就是这里了，如表16-6所示。

表16-6 下一种情况的分析矩阵——加拿大销售

	美国销售	加拿大销售
计算运费	按照 UPS 费率	
验证地址	使用美国邮政规则	
计算税额	使用美国和当地税收规则	
金额	美元	加拿大元

重复这一过程，直到填完如表16-7所示的表格。

表16-7 下一种情况的分析矩阵——加拿大销售

	美国销售	加拿大销售
计算运费	按照 UPS 费率	按照联邦快递费率
验证地址	使用美国邮政规则	使用加拿大邮政规则
计算税额	使用美国和当地税收规则	使用 GST 和 PST
金额	美元	加拿大元

下一种情况的完整矩阵如表16-7所示。

当然，事情并非总是如此顺利。新情况往往会带来新功能。但是，这是好事情。这样我们就能够有机会检验分析的完整性。我与客户交流的经验表明，他们通常无法提供完整的需求，因为他们总是按正常情况考虑问题，忽视异常情况（而我们却是必须要处理的）。

……在分析过程中寻找不完整和不一致

在构建矩阵的过程中，我发现了需求中的遗漏。我将使用这些信息扩展分析。这些不一致说明客户提供的信息不完整。也就是说，在某种情况下某个顾客可能提某种特殊需求，而另一个顾客则不会。例如，在来自美国市场的需求中，没有提到最大重量的问题，而加拿大市场可能有最重31.5千克的限制。通过比较这些需求，我又找到美国客户的联系人，专门询问她重量限制的问题（实际上可能并不存在此限制），然后补上这一漏洞。

3.有了新的概念就扩展分析矩阵

随着时间流逝，我们总是需要处理新的情况（例如业务扩展到了德国）。当你发现了某种情况中存在一个新概念时，在矩阵中增加一行，即使它并不适用于其他情况，如表16-8所示。

表16-8 扩展分析矩阵

	美国销售	加拿大销售	德国销售
计算运费	按照 UPS 费率	按照联邦快递费率	按照德国货运公司费率
验证地址	使用美国邮政规则	使用加拿大邮政规则	使用德国邮政规则
计算税额	使用美国和当地税收规则	使用 GST 和 PST	使用德国增值税
金额	美元	加拿大元	欧元
日期	mm/dd/yyyy	mm/dd/yyyy	dd/mm/yyyy
最大重量			30 千克

美国和加拿大有最大重量吗？可能没有；也可能有，但是客户忘了提到这一点。现在，有一个很好的具体问题要问了。我的客户对于回答具体问题是非常擅长的，而对于“还有别的什么吗？”这样的问题，他们一般并不擅长。

关于客户的一点说明

与客户打交道的经验使我懂得以下几点。

他们通常非常了解他们的问题域（大多数我永远也赶不上）。

一般情况下，他们不会像开发人员经常的那样在概念层次表达事情。相反，他们会谈得非常具体。

他们经常用“总是”表达“通常”。

他们经常用“从不”表达“很少”。

总之，对于非常具体的问题，客户详细的回答一般是可信的，但是他们一般性的回答却不可信。我尝试在非常具体的层次与他们沟通。即使是那些听上去似乎是在概念层次考虑问题的客户，往往并非真的如此，只是想努力帮助我而已。

4.用行发现规则

现在概念已经找到，对这些已知信息应该怎么处理呢？我怎样着手实现？

观察表16-8中的矩阵。第一行标记为“计算运费”，包括“按照UPS费率”、“按照联邦快递费率”和“按照德国货运公司费率”。这一行表示：

实现“计算运费费率”的一般规则；

必须实现的具体规则集——也就是在不同国家（地区）使用的货运公司。

实际上，每一行都表示实现一个一般规则的特定方式。其中的两行（金额和日期）可以在对象层次上处理。例如，金额可以用包含货币对象的对象来处理。许多计算机语言的库中都支持不同国家（地区）的不同日期格式。表16-9说明了处理每一行的概念性的方式。

表16-9 具体实现规则：行

	美国销售	加拿大销售	德国销售
计算运费	计算运费费率的各种方式的具体实现		
验证地址	验证地址的各种方式的具体实现		
计算税额	计算应付税额的各种方式的具体实现		
金额	使用包含 Currency 字段和 Amount 字段的 Money 对象，可以自动兑换货币		
日期	使用包含 display 方法的 Data 对象，可以根据顾客所在国家（地区）的要求显示日期		
最大重量	最大重量的各种方式的具体实现		

5.用列发现实现

那么列又表示什么呢？它们是针对列所表示的情况的特定实现。如表16-10所示。

表16-10 具体实现规则：列

	美国销售	加拿大销售	德国销售
计算运费 验证地址 计算税额 金额 日期 最大重量	当我们有美国顾客时，使用这些实现	当我们有加拿大顾客时，使用这些实现	当我们有德国顾客时，使用这些实现

例如，第一列说明了用于处理美国销售订单的具体实现。

5a.从这种分析中确定模式：观察行

应该怎样把这些深入认识转化成模式呢？再对表 16-8 进行观察，每一行都表示实现最左列中概念的特定方式。例如：

在“计算运费”行中，“按照UPS费率”；“按照联邦快递费率”两项实际上表示“应该怎样计算运费”；所封装的算法是“运费费率计算”。具体的规则将是“UPS费率”、“加拿大费率”和“德国费率”；

下两行也是由不同规则及其相关具体实现组成的；

“金额”和“日期”两行表示可能在整个应用程序中保持一致的类，但是会根据国家（地区）的不同表现不同。

因此，除“金额”和“日期”两行之外的行可以考虑运用Strategy模式，如表16-11所示。例如，第一行的对象可以实现为封装了“计算运费”规则的Strategy模式。

表16-11 用Strategy模式实现

	美国销售	加拿大销售	德国销售
计算运费	这一行的对象可以用封装“计算运费”规则的 Strategy 模式实现		
验证地址	这一行的对象可以用封装“验证地址”规则的 Strategy 模式实现		
计算税额	这一行的对象可以用封装“计算税额”规则的 Strategy 模式实现		
金额	使用包含 Currency 字段和 Amount 字段的 Money 对象，可以自动兑换货币		
日期	使用包含 display 方法的 Data 对象，可以根据顾客所在国家（地区）的要求显示日期		
最大重量	这一行的对象可以用封装“计算最大重量”规则的 Strategy 模式实现		

5b.从这种分析中确定模式：观察列

以类似的方式，再来观察列。每一列描述了每种情况用哪一个规则。这些项表示该情况需要的对象系列。这听上去像是Abstract Factory模式，如表16-12所示。

表16-12 用Abstract Factory模式实现

	美国销售	加拿大销售	德国销售
计算运费 验证地址 计算税额 金额 日期 最大重量	这些对象可以通过使用 Abstract Factory 模式协调	这些对象可以通过使用 Abstract Factory 模式协调	这些对象可以通过使用 Abstract Factory 模式协调

6.得到高层设计

某些行表示Strategy模式，每一列都表示Abstract Factory模式中的一

个对象系列，有了这些信息，现在可以得到一个高层应用程序设计了，如图16-1所示。

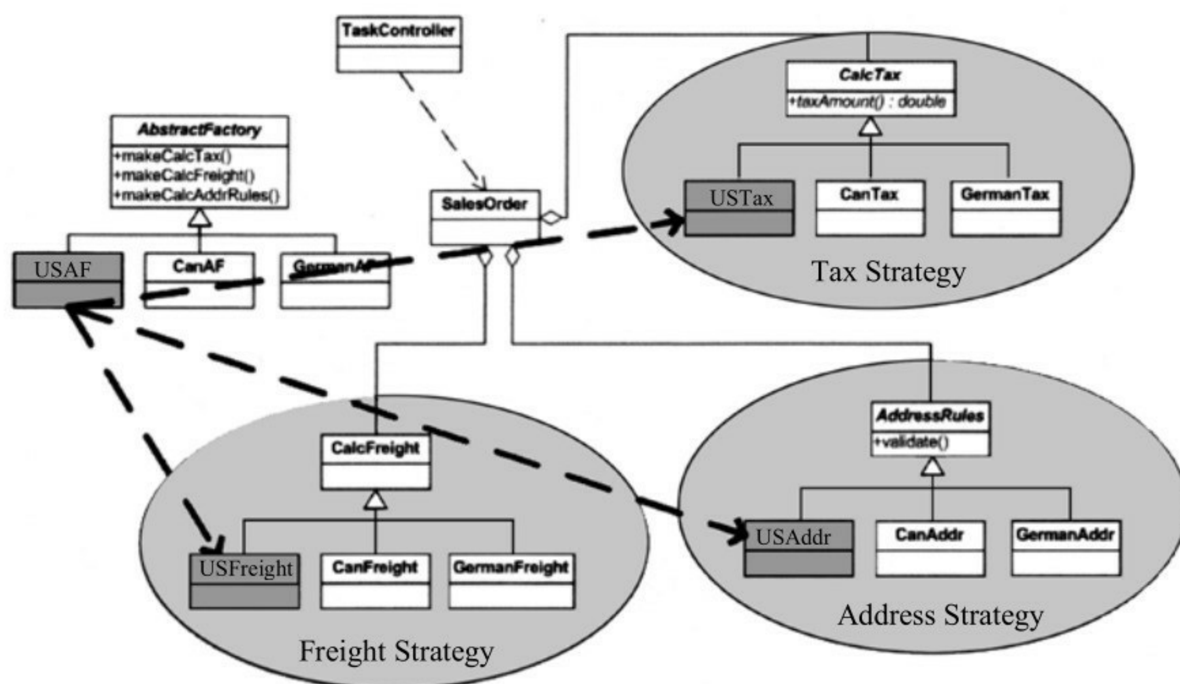


图16-1 高层应用程序设计

16.4 实践注记

其他模式

在实践中，几乎任何模式都涉及分析矩阵中可能存在的多态性。我曾经在分析矩阵中使用过的其他模式还包括 Bridge、Composite、Chain of Responsibility、Command、Decorator、Iterator、Mediator、Observer、Proxy、Template和Visitor。

图16-2所示为一个序列图，说明了美国情况中的工作原理。

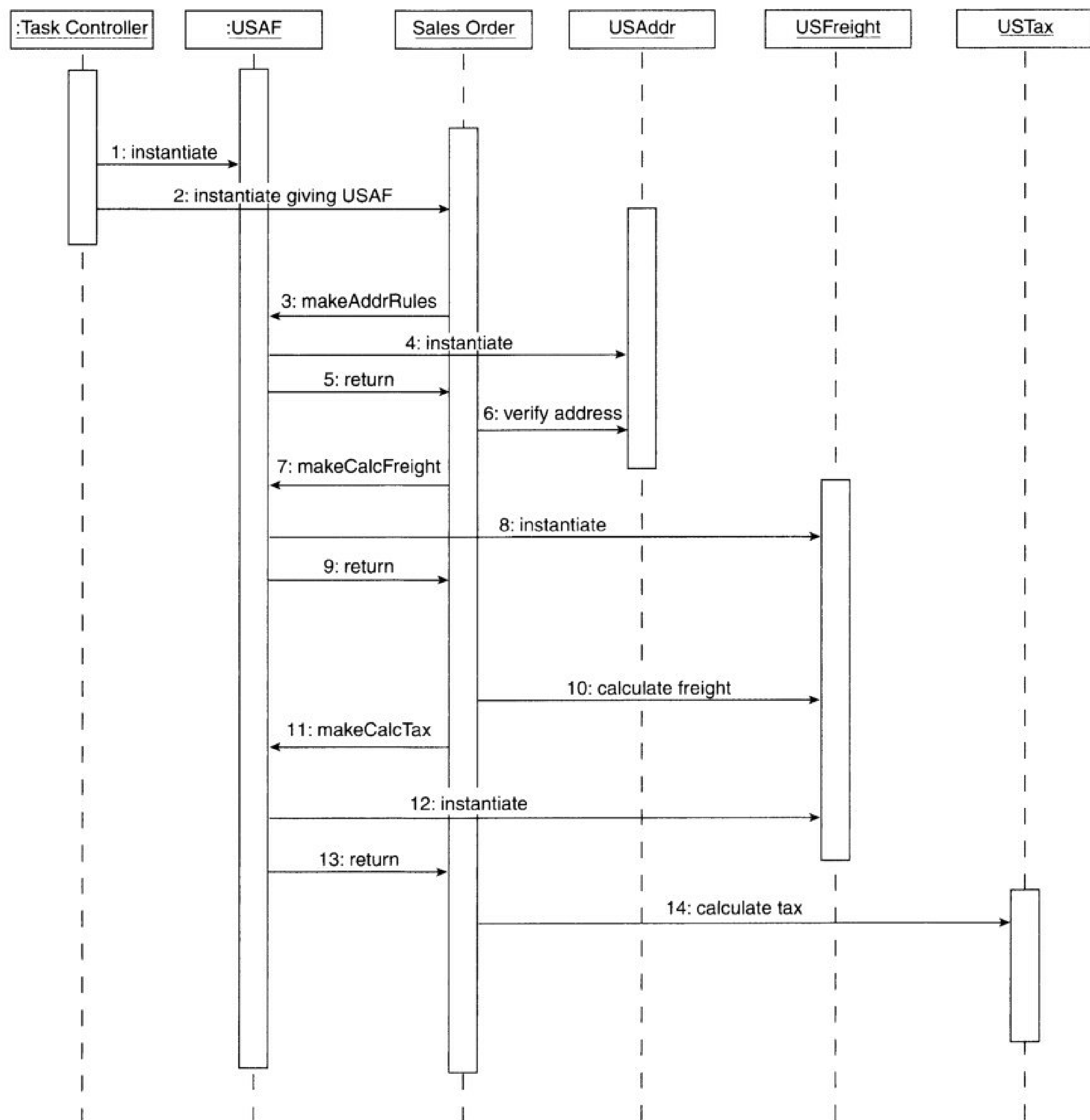


图16-2 美国情况的序列图

例如，如果在我们的电子商务系统中加入了打印销售票据的需求，并发现如下变化。

美国销售票据需要表头。

加拿大销售票据需要表头和页脚。

德国销售票据需要两个不同的页脚。

我将在另一行中包含这些信息，其中每项与一种销售票据的格式相关。这一行的需求将使用 Decorator 模式来实现，这样就能比较容易地改变所用的页脚和表头（以及它们的顺序）。

分析矩阵的适用性

尽管分析矩阵很少能捕获一个特定问题域的所有方面，但我发现它至少可以用于大多数问题域。我发现，它在所提供的特殊情况太多，我的脑子无法得到总体视图时最有用。

问题越大，分析矩阵越有用

情况通常比这还要糟糕。需求的各种不同情况很少会以非常协调的形式陈述给分析师或开发人员。但是，这不会明显地使分析矩阵的过程更复杂。在这种情况下，对于一个特性，查看最左一列，看这个特性是哪个概念的变化。如果找到了这个概念，就将该概念放入那一行中。如果找不到这样的概念，就说明必须创建新行。

在极端情况下，分析矩阵可能是唯一的处理事情的方法。我曾经有一个存在大量特殊情况的客户端。每种情况都是一个分别开发的文档控制系统。问题是要将这些文档控制系统集成起来。特殊情况如此之多（因此矩阵也有许多行），同时考虑整个问题是不可能的。分析师对其中涉及的所有东西无法在概念上很好地把握，他们只是谈论一般规则和例外情况。通过分别考虑每种情况，可以抽象出公共数据和行为（显示在最左一列中），然后通过设计模式来实现它们。

经常会用到子矩阵

在实践中，一个简单的矩阵往往是不够的。即使在我们的这个简单案例中，也很容易想象，如果一个国家（地区）中有超过一种货运方式会变成什么样子。在主矩阵中再有子矩阵有时是可取的。图16-3给出了一个例子。

	美国销售	**加拿大销售**
计算运费	使用美国费率	使用加拿大
验证地址	使用美国邮政规则	使用加拿大邮政规则
计算税额	使用州和当地税务规则	使用 GST 和 PST
金额	美元	加拿大元

计算运费	**使用美国邮政**	**UPS**	**联邦快递**
验证地址	使用美国邮政规则	使用美国邮政规则（无邮箱）	使用美国邮政规则
上门取货费	不适用	$5.00	$4.00
超重限额	50 磅	70 磅	无限制

图16-3 矩阵中的矩阵

实际上我只在无法直接得到我称为“共性与可变性表”的时候，才使用矩阵中的矩阵。图16-3中的内嵌矩阵对应的表如表16-13所示。

表16-13 内嵌矩阵定义的共性与可变性

计算运费共性	取货附加费共性
UPS 费率	无
USPS 费率	5 美元
联邦快递费率	4 美元
验证地址共性	**最大重量共性**
美国邮政规则	50 磅
美国邮政规则（无邮箱时）	70 磅
	无最大限制

16.5 小结

本章内容

概念中的变化可能是分析师所面临的最大挑战之一。本章中介绍了一种简单的分析工具，我发现它有助于我们弄清这些变化的意义。我将这种工具称为分析矩阵，它是以Christopher Alexander和Jim Coplien所提出的概念为基础的。我将这种工具应用到一个示例问题，说明了它怎样展现出问题中本身存在何种模式。虽然这种工具对封装变化很有用处，而且有助于思考问题域，但我并没有夸大其词，说它能捕获设计的所有特性。

复习题

简答题

1.分析矩阵的最左一列是什么？

2.分析矩阵的行表示的是什么？

3.分析矩阵的列表示的是什么？

4.本书所讲述的哪些模式可以出现在分析矩阵中？

阐述题

1.分析矩阵在哪一层次视角发挥作用？

2.分析矩阵在哪方面与共性/可变性分析类似？

观点与应用题

1.模式能够有助于更有效地处理变化吗？

2.你是否同意本章中对用户的评论？能够从亲身经历中给出例子来说明吗？

3.你相信分析矩阵对大多数问题领域都普遍适用吗？

第17章 Decorator模式

17.1 概览

本章内容

本章继续对第9章开始的电子商务案例进行研究。

在本章中，我们将：

描述该案例的一个新需求：为打印的销售票据添加表头、页脚信息；

说明Decorator模式如何灵活地处理这一需求；

讨论如何用Decorator模式处理输入/输出（尤其是Java的I/O）；

给出Decorator模式的关键特征；

描述我在实践中使用Decorator模式的一些经验；

描述Decorator模式的本质不是一个链表，而是一组可选的装饰对象。

17.2 更多细节

扩展类图

图9-2所示为案例研究的基本结构。图17-1更详细地显示了这一结构。其中展示了SalesOrder对象使用一个SalesTicket对象打印销售票据。

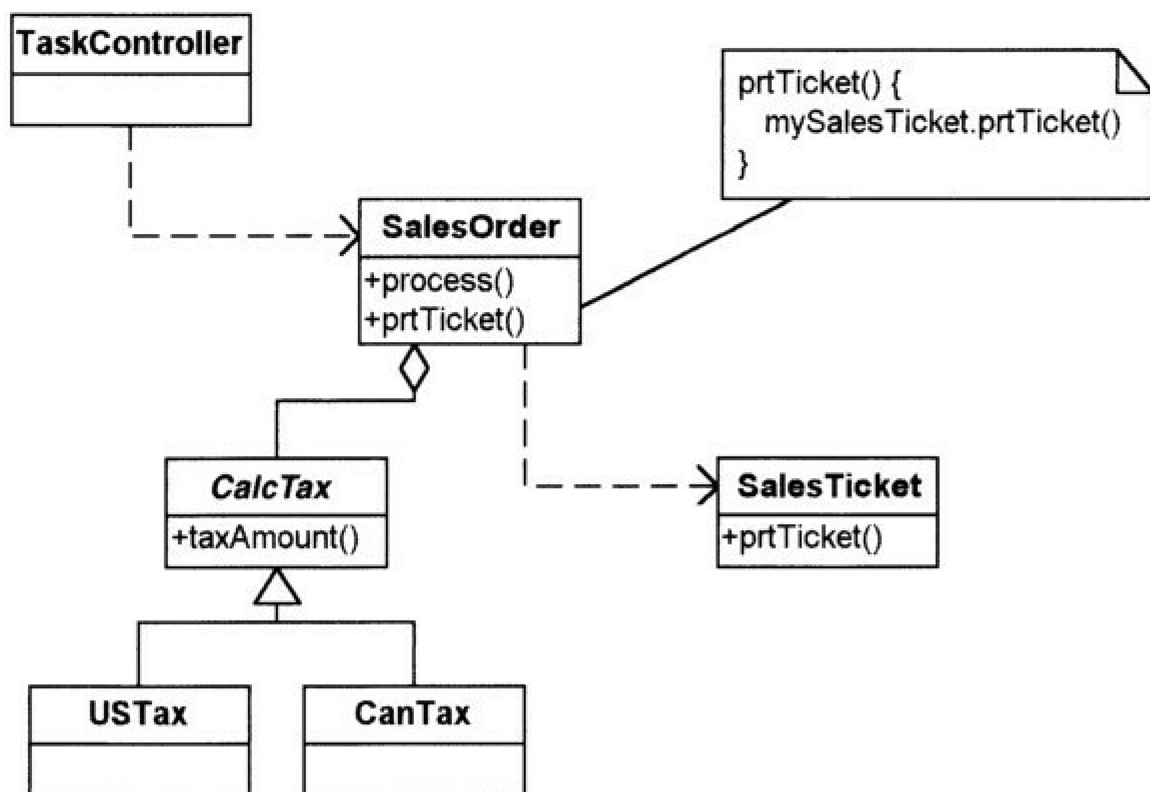


图17-1 SalesOrder对象使用SalesTicket对象

在第9章已经看到，SalesOrder对象使用一个CalcTax对象计算订单的税额。为了实现打印功能，SalesOrder调用SalesTicket对象，请求它打印票据。这是一个不错的、合理的模块化设计。

新需求：添加表头

在编写这个应用程序的过程中，假设我接到一个新的需求：为SalesTicket对象添加表头信息。

一种方式：在Sales-Ticket类中使用switch语句

怎样处理这一新需求呢？如果编写的程序只供一家公司使用，可能直接在 SalesTicket 类中添加对表头和页脚的控制最简单，如图17-2所示。

这种办法不够灵活

在这一解决方案中，控制信息在 SalesTicket类里，有标志说明是否

需要打印表头或页脚。

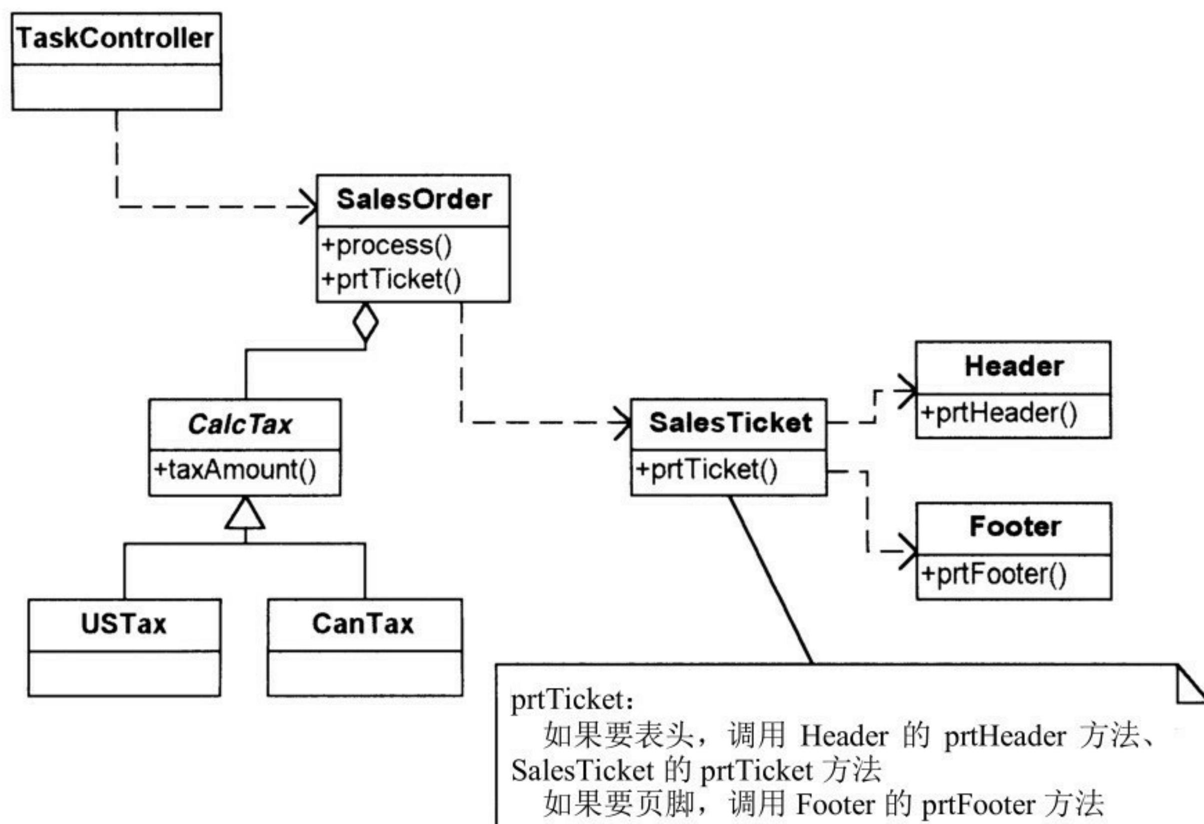


图17-2 SalesOrder对象使用SalesTicket对象，有各种不同的选择

如果无需处理很多选择，或者如果销售订单使用的表头不会变，这一方案将行之有效。

如果必须处理很多不同类型的表头和页脚，每次只打印一种类型，那么我可以考虑为表头使用一次 Strategy 模式，为页脚再使用一次Strategy模式。

如果必须一次打印一个以上的表头或页脚，情况会怎样？如果表头或页脚的顺序需要改变，又会怎样？各种组合的数量会迅速将我们

Decorator 模式的用武之地

在这种情况下，可以证明Decorator模式非常有用。Decorator模式并不通过一个控制方法控制新增功能，而是建议以需要的正确顺序将所需功能串联起来，进行控制。Decorator模式将这样一个功能链的动态构建

与使用功能的客户（本案例就是SalesOrder对象）分离开来。而且还将功能链的构建与链组件（比如表头、页脚和SalesTicket）分离开来。这样就能灵活地使用这些组件。

17.3 Decorator模式

意图，《设计模式》一书的说法

《设计模式》一书中对Decorator模式的意图是这样叙述的：

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式比生成子类更为灵活。[8]

工作原理

Decorator模式的工作原理是：可以创建始于Decorator对象（负责新功能的对象）终于原对象的一个对象“链”，如图17-3所示。

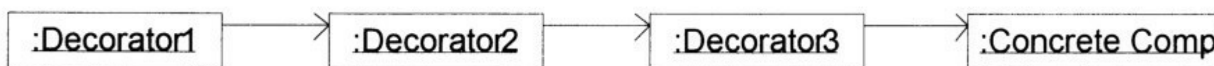


图17-3 Decorator链

Decorator 模式是一个对象链

图17-4中Decorator模式的类图隐含了图17-3所示的对象链。每条链都始于一个Component对象（ConcreteComponent或Decorator）。每个Decorator对象后面都跟着另一个Decorator对象或原Concre-teComponent对象。对象链总是终于一个ConcreteComponent对象。

例如，在图17-4中，ConcreteDecoratorB对象执行其 Opera-tion方法，然后调用Decorator类的Operation方法。这又将调用ConcreteDecoratorB对象之后的Component对象的Operation方法。

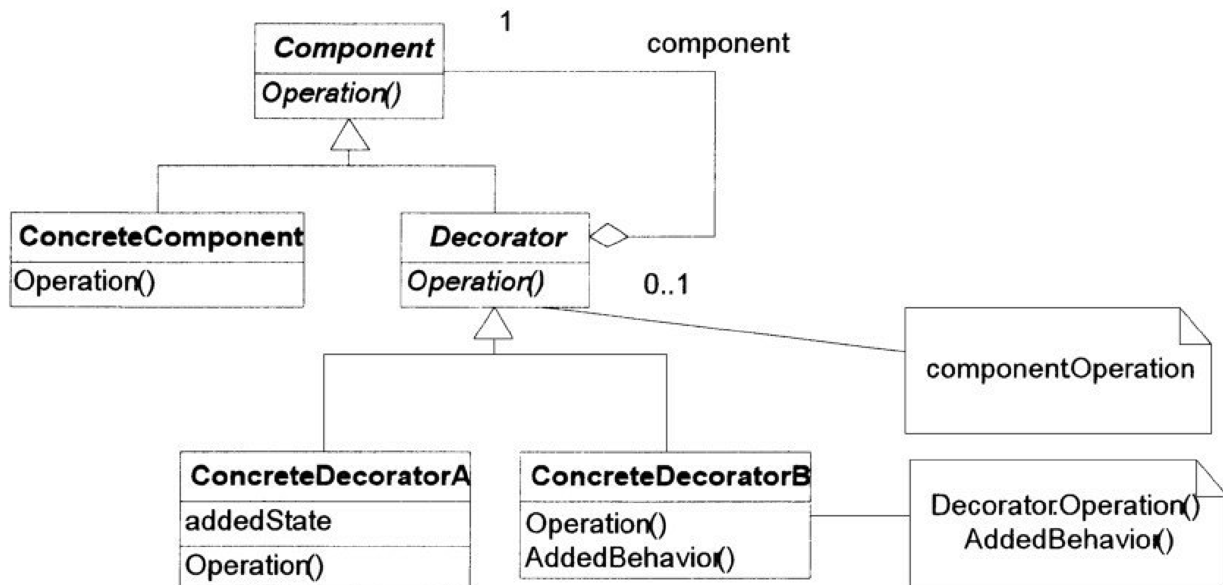


图17-4 Decorator模式的类图

17.4 将Decorator模式应用到我们的案例研究

在本案例中

在此案例中，SalesTicket对象就是 ConcreteComponent，具体的装饰则是表头和页脚。图17-5说明了Decorator模式在这个案例中的应用。

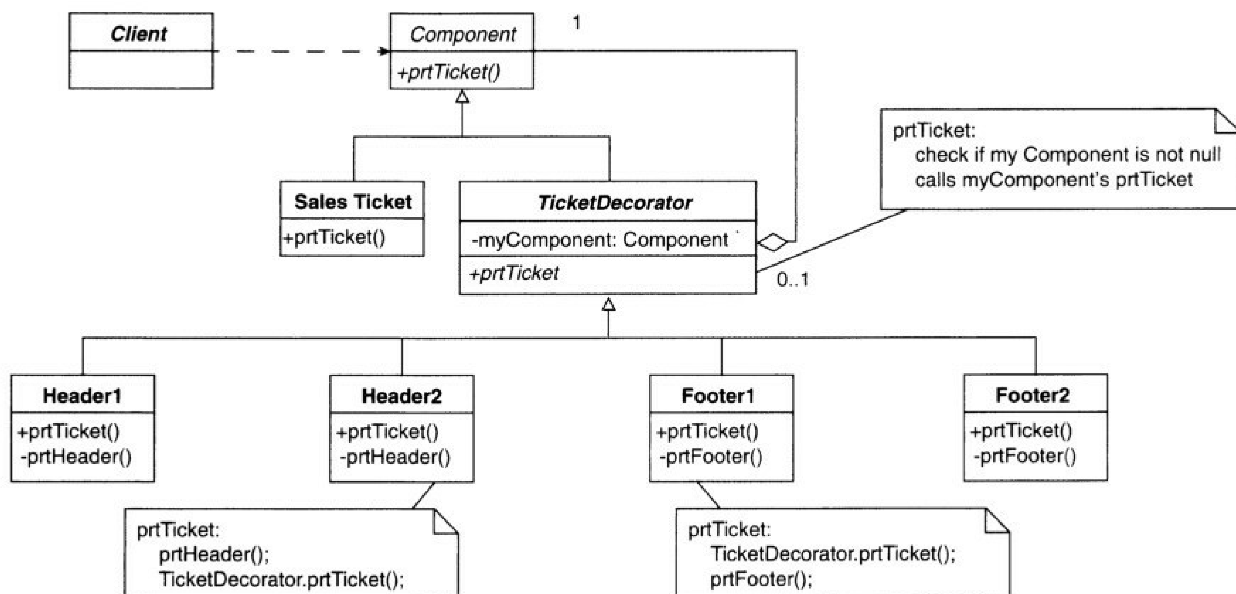


图17-5 设置表头和页脚，让票据看起来像一份报表

实现了的模式

图 17-6 说明了 Decorator模式在“一个表头一个页脚”的情况下的应用。

如何运作：Decorator封装其后的对象

每个 Decorator 对象都对其后紧跟的对象封装自己的新功能。每个Decorator对象在被装饰的功能之前（对于表头）或之后（对于页脚）或者与前两者同时执行自己的附加功能。要了解其中如何运作，最简单的办法就是看一看某个具体例子的代码并通读一遍。我们来看例17-1。

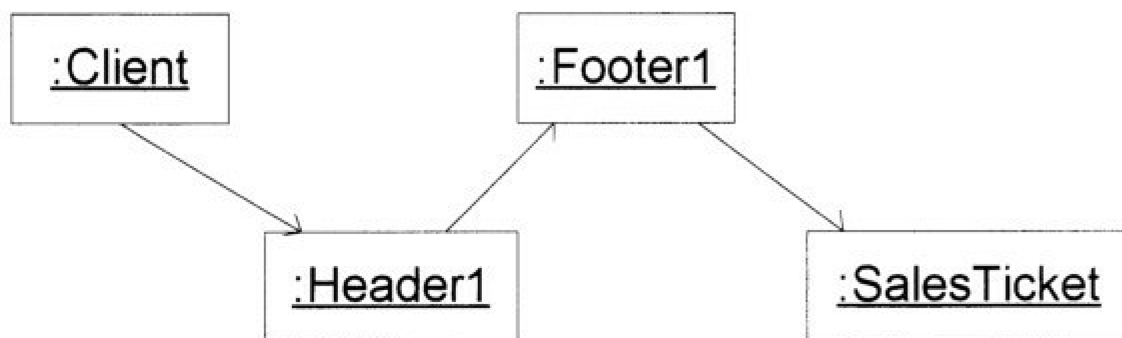


图17-6 Decorator模式示例类图

例**17-1 Java**代码片段：**Decorator**模式

```
public class Client {
  public static void main( String[] args) {
      Factory myFactory;
         myFactory= new Factory();
         Component myComponent=
            myFactory.getComponent();
   }
}
abstract public class Component {
   abstract public void prtTicket();
}
public class SalesTicket extends Component {
   public void prtTicket() {
      // 这里是打印销售票据的代码
   }
}
abstract public class TicketDecorator extends
Component {
   private Component myTrailer;
   public TicketDecorator (Component myComponent) {
      myTrailer= myComponent;
   }
   public void callTrailer () {
      if (myTrailer != null) myTrailer.prtTicket();
   }
}
public class Header1 extends TicketDecorator {
   public Header1 (Component myComponent) {
      super( myComponent);
   }
   public void prtTicket () {
      // 这里是打印表头 1 的代码
      super.callTrailer();
   }
}
public class Header2 extends TicketDecorator {
   public Header2 (Component myComponent) {
      super( myComponent);
```

```
    }
    public void prtTicket () {
        // 这里是打印表头 2 的代码
        super.callTrailer();
    }
}
public class Footer1 extends TicketDecorator {
    public Footer1 ( Component myComponent) {
        super( myComponent);
    }
    public void prtTicket() {
        super.callTrailer();
        // 这里是打印页脚 1 的代码
    }
}
public class Footer2 extends TicketDecorator {
    public Footer2 ( Component myComponent) {
        super( myComponent);
    }
    public void prtTicket() {
        super.callTrailer();
        // 这里是打印页脚 2 的代码
    }
}

public class Factory {
    public Component getComponent () {
        Component myComponent;
        myComponent= new SalesTicket();
        myComponent= new Footer1( myComponent);
        myComponent= new Header1( myComponent);
        return myComponent;
    }
}
```

如果我需要这样的一张销售票据：

代码分析

HEADER l

SALES TICKET

FOOTER l

那么myFactory.getComponent 将返回

return(new Header1(new Footer1 (new

SalesTicket()));

这创建了一个 Headerl 对象，其后是一个 Footerl 对象，再后是一个SalesTicket对象。

如果我需要这样的一张销售票据：

HEADER l

HEADER 2

SALES TICKET

FOOTER l

那么myFactory.getComponent 将返回

return(new Header1(new Header2 (new Footer1(

new SalesTicket()))));

这创建了一个 Headerl 对象，跟着是一个 Header2 对象，其后是一个Footerl对象，再后是一个SalesTicket对象。

根据责任进行分解

Decorator模式帮助我们将问题分为以下两部分。

如何实现提供新功能的对象。

如何为每种特殊情况组织对象。

这样能够将Decorator对象的实现与决定如何使用Decorator的对象分离开来，从而提高了内聚性，因为每个 Decorator 对象只用关心自己添加的功能——无需关心自己如何被添加到对象链中。这还使我们能够任意地重排Decorator的顺序，无需改变其的任何代码。

17.5 另一个例子：输入/输出

流输入/输出

Decorator 模式的一个常见应用场合是流输入/输出。在讨论如何在该场合使用Decorator模式之前，我们先来了解一些流输入/输出的知识。讨论将仅限于“输入”，因为输出的工作方式与输入类似（了解了一个方向的工作原理，可以推知另一个方向）。对于任何特定的流输入，有且只有一个数据源，但可以有任意数量（包括0个）在输入流上执行的操作。例如，可以从下列数据源读取数据：

文件；

套接字，然后解码输入流；

文件，然后解压缩输入数据；

字符串；

文件，解压缩输入，然后解码。

根据数据发送（或存储）方式的不同，以上任何行为的组合都有可能。可以这样考虑：任何源都可以被任何行为的组合装饰。表 17-1 列出了流输入的一些可能情况。

表17-1 数据源和行为的种类

数 据 源	行　为
字符串	缓冲输入
文件	计算校验和
套接字（TCP/IP）	解压缩
串行端口	解码（任意多种方法）
并行端口	选择筛选（任意多种方法）
键盘	

语言反映了这一点

使用面向对象语言的开发人员可以利用这一点，从一个公共抽象类派生数据源对象和行为对象。每个行为对象可以在构造函数中获得数据源或前一个行为。然后在这些对象实例化时，就构建出一个行为链（每

个行为对象都获得指向其后紧跟的对象的引用）。数据源对象派生自Concrete-Component类（见图17-4），行为对象则是装饰。请注意，现在Concrete-Component这个词有些名不副实，因为它现在是抽象类。

例如，为了得到这样的行为："从文件中读取，解压缩输入，然后解码"，可以按以下步骤进行。

1.按照以下步骤构建装饰链。

a.实例化一个文件对象。

b.将文件对象的引用传递给解压缩对象的构造函数。

c.将解压缩对象的引用传递给解码对象的构造函数。

2.读取、解压缩并解码数据——所有这些对于使用它的客户对象都是不可见的。客户对象只知道自己有某种输入流对象。

如果客户需要从另一个数据源获得输入，在生成装饰链的过程中就实例化另一个数据源对象，使用的行为对象仍相同。

理解"应有尽有的流动物园"[9]

Java 因为其令人迷惑的多种流输入及相关的类而臭名昭著。在Decorator 模式的背景下，理解这些类要容易得多。这些类直接派生自java.io.InputStream（ByteArrayInputStream、FileInputStream、FilterInputStream、InputSteam、ObjectInputSteam、Sequence-InputStream和 StringBufferInputStream），都扮演被装饰对象的角色。所有的装饰都（直接或间接地）派生自FilterInputStream类。

理解了Decorator模式，就可以解释为什么Java要求这些对象一个封装在另一个里——这使程序员能够从可获得的不同的行为中选取任意数量的组合。

17.6 实践注记：使用Decorator模式

实例化链

Decorator模式要求对象链的实例化与使用它的Client对象完全分离开。最典型的实现是通过使用工厂对象，根据某些配置信息实例化对象链。

用于测试的 Decorator模式

我曾经使用 Decorator 模式封装对一个被测对象的前置条件和后置条件测试，结果很令人满意。在测试期间，链中的第一个对象可以在调用紧跟其后的对象之前进行全面的前置条件测试。在调用了其后紧跟的对象之后，这个对象又可以立即全面地测试后置条件。如果我将在不同时刻进行不同的测试，可以将每个测试放在不同的 Decorator 中，然后根据需要的测试组合将它们串联起来。

Decorator模式：关键特征

意图

动态地给一个对象添加职责。

问题

要使用的对象将执行所需的基本功能。但是，可能需要为这个对象将添加某些功能，这些附加功能可能发生在对象的基础功能之前或之后。请注意，Java基础类在I/O处理中广泛使用了Decorator模式。

解决方案

可以无需创建子类，而扩展一个对象的功能。

参与者与协作者

ConcreteComponent让Decorator对象为自己添加功能。有时候用ConcreteComponent的派生类提供核心功能，在这种情况下Concrete-Component类就不再是具体的，而是抽象的。Component类定义了所有这些类所使用的接口。

效果

所添加的功能放在小对象中。好处是可以在ConcreteComponent对象的功能之前或之后动态添加功能。注意，虽然装饰对象可以在被装饰对象之前或之后添加功能，但对象链总是终于ConcreteComponent对象。

实现

创建一个抽象类来表示原类和要添加到这个类的新功能。在装饰类中，将对新功能的调用放在对紧随其后对象的调用之前或之后，以获得正确的顺序。

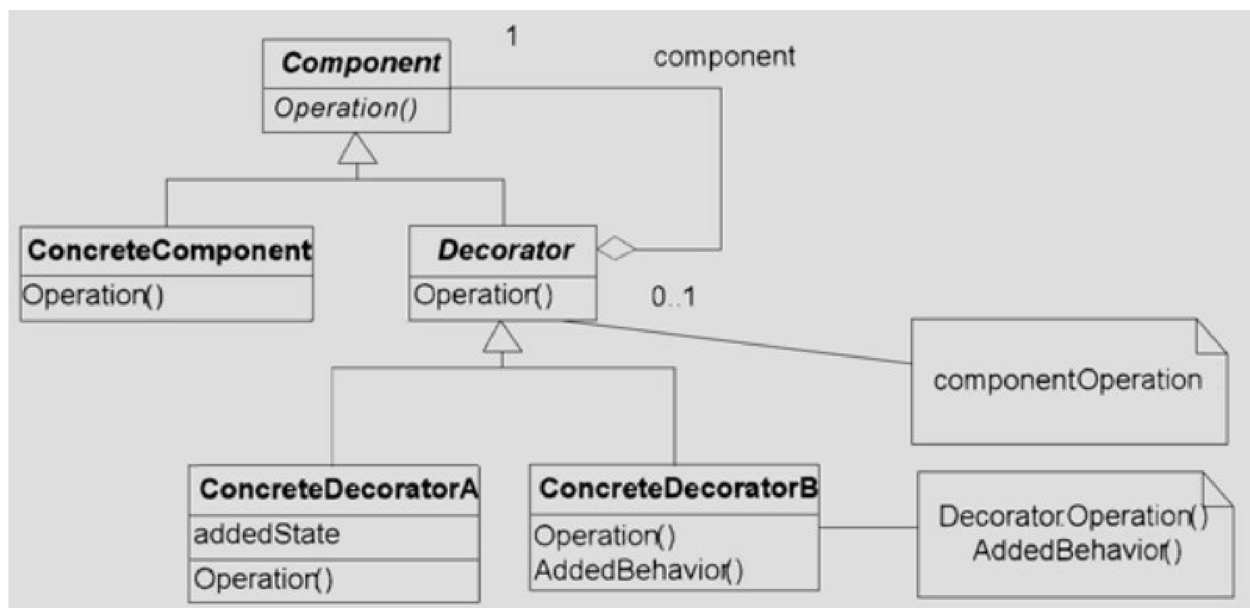


图17-7 Decorator模式的通用结构图

17.7 Decorator模式的本质

结构本身并非模式

Decorator模式的适用场合是，各种可选的功能在另一个肯定要执行的功能之前或者之后执行。例如，在发送一条信息（transmission）之前（这是通常要执行的），可能需要加密、压缩、转换数据校验码或者以

任何顺序执行的任意数量的其他工作。怎样做才算最佳方案呢？Decorator 模式的实现告诉我们，构建一个止于Transmission对象的可选功能链表。装饰对象应该与Transmission对象有相同的接口。

这种实现实际上是一种很糟糕的设计。例如，假设这些“装饰对象”（即可选的功能）是由不同开发组开发的，而且系统可能抛出一些异常，需要由每个装饰对象进行处理。理想情况下，可以相信这些开发组都能够按预期地实现——正确地捕获这些异常。然而，如果他们没有做到会怎样呢？如果一个异常抛出了，而某个开发组编写的代码无法捕获它情况又会怎样呢？整个系统这时可能轰然倒塌。

另一种方案是让客户对象捕获异常。但是，这也就丧失了Decorator模式的价值，因为客户类现在需要完成比它以前更多的任务。

更健全的解决方案是实现一个与Transmission对象接口相同的对象集合。它调用装饰对象，捕获后者没有捕获的任何必需捕获的异常。事实上，这可能具有附加的好处：装饰对象再也不必要与Transmission对象接口相同了。

关键在于，要看到Decorator模式有如下约束因素。

存在几种可选的功能。

这些装饰对象可能遵循也可能不遵循所有规则。

需要某种方式以所需的不同顺序调用这些装饰对象，但是又不能加重客户对象的负担。

不希望应用程序必须承担知道使用哪些装饰对象（甚或是否存在）的职责。

这样思考Decorator模式将使模式的意图和实现分离开来。

17.8 小结

本章内容

Decorator 模式是为已有功能动态地添加更多功能的一种方式。实践中，必须创建一个对象链提供所需的行为。链中的第一个对象由Client对象调用，Client 对象无需关心对象链的创建。使对象链的创建与使用保持独立，这样Client对象就不会受添加功能的新需求的影响。

复习题

简答题

1.每个Decorator对象封装的是什么？

2.Decorator对象的两个经典例子是什么？

阐述题

1.Decorator模式是怎样有助于分解问题的？

2.“结构本身并非模式。”这个结论是在对Decorator模式本质的讨论中得出的。这是什么意思？为什么这一点很重要？

观点与应用题

1.你认为为什么《设计模式》一书称此模式为“Decorator”？就它的功能而言这个名字合适吗？给出你的理由。

2.有时候人们会将模式当作解决问题的处方。这种观点有什么错误？

---

[1].这些怪怪的名字可不是我取的，但是请相信我，它们比听上去要容易理解。

[2].Meyer B.，Object-Oriented Software Construction，Upper Saddle River，NJ:Prentice Hall，1997，p.57。

[3].参见本书配套网站 http://www.netobjectives.com/dpexplained，其中有Robert C.Martin的妙文The Open-Closed Principle的链接。（此处开放对应的更确切的中文意义是无限制、允许，封闭是不允许、限制。

[4].Martin B.，Agile Software Development:Principles,Patterns,and Practices，Upper Saddle River，N.J.: Prentice Hall，2003，p.127。

[5].Liskov Barbara，Data Abstraction and Hierarchy,SIGPLAN Notices，23.5(May 1988)。

[6].对此问题更加深入的讨论，请参考Alan Shalloway的文章Can Patterns Be Harmful?（模式可能有害吗？），可以在本书配套网站http://www.netobjectives.com/dpexplained找到。这篇文章也非常适合想初步了解何谓设计模式的读者。

[7].我相信这种方法与泛型编程和面向方面编程的关系都非常紧密。但是，这些主题超出了本书的范畴。请参考第25章获得这些领域的更多信息。

[8].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Eelements of Reusable Object-Oriented Software，Boston，Mass.:Addison-Wesley，1995，p.175。

[9].Horstmann C.，Core Java Volume 1 Fundamentals,Palo Alto，CA:Pearson Education，1999，p.627。

第六部分 其他重要模式

概览

本部分内容

本部分将通过说明在功能之外可能出现的不同变化类型，扩展模式封装变化的概念。

章 讨论的主题

18 Observer模式

如何封装“哪些对象需要得到通知”和“在什么情况下得到通知”方面的变化。

19 Template Method模式

如何消除冗余，封装应该完成的过程。也就是说，如何处理必须采取的步骤实现中的变化。

第18章 Observer模式

18.1 概览

本章内容

本章将继续讨论由第9章开始，第16章又接着讨论的电子商务案例研究。在本章中，我们将：

介绍模式的分类；

通过讨论案例中的更多需求来介绍Obsever（观察者）模式；

将Observer模式应用到我们的案例中；

描述Observer模式；

给出Observer模式的关键特征；

描述我在实践中使用Observer模式的一些经验。

18.2 模式的分类

《设计模式》一书分了3类

需要掌握的模式很多。为了帮助读者，《设计模式》一书将这些模式组织成三个通用类别，如表18-1所示。[1]

表18-1 模式的类别

分　类	意　图	本书中的例子	用　途
创建型	创建或实例化对象	Abstract Factory 模式（第 11 章） Singleton 模式（第 21 章） Double-Checked Locking 模式（第 21 章） Factory Method 模式（第 23 章）	实例化对象
结构型	将已有的对象组合起来	Facade 模式（第 6 章） Adapter 模式（第 7 章） Bridge 模式（第 10 章） Decorator 模式（第 17 章）	处理接口 将实现与抽象联系起来
行为型	给出一种提供灵活（变化）行为的方式	Strategy 模式（第 9 章）	封装变化

说明：关于 Bridge模式和Decorator模式的分类

刚开始学习设计模式时，我非常惊讶地看到《设计模式》一书居然将Bridge　和Decorator　模式归为结构型模式而不是行为型模式。毕竟，它们看上去是用来实现不同行为的。原来，我根本就没有理解《设计模式》一书的分类方式。结构型模式的作用是将已有的功能组合起来。在Bridge模式中，我们通常从抽象和实现开始，然后用Bridge模式将它们组合起来。在Decorator模式中，是希望用更多附加的功能对原有的功能类进行装饰。它们的作用都是组合功能，所以是结构型的。

实践中，结构型模式所用的对象开始时都是分离的。但是我发现，许多时候模式都是用于按我考虑模式的方式会将这些对象联系起来的时候。本书前面讲述的CAD/CAM问题就是这样的一个例子。在该例中，Bridge　模式在发现“两个实现与使用它们的特征分离”这一事实时发挥了重要作用。 我发现，Bridge模式在提醒我将类中纠缠的事物分离这方面非常有用。

我的“第4个”类别：解耦型模式

我发现增加第4个模式类别很有价值，这一类别的主要目的是将一个对象与另一个对象解耦。这类模式的一个动机是允许伸缩或增加灵活性，我称这一类模式为解耦型模式。因为大多数解耦型模式都属于《设计模式》一书中行为型类别，差不多也可以将它们称为行为型类别的子集。我选择增加第4个类别，只是因为本书的意图是反映我如何看待模式，注意力应该放在动机上——在这里，这一类的动机就是解耦。

我不想在分类的理由上着墨太多。分类的意义在于深入考察模式的作用。事实上，大多数模式都是这四种特质的组合。

Observer 模式是一个解耦型（行为型）模式

本章将讨论Observer模式，这是我所能想到的“解耦型模式”最佳范例。《设计模式》一书将Observer归为行为型模式。

18.3 国际电子商务案例的更多需求

新需求：为新消费者进行操作

在编写应用程序的过程中，假设又来了一个新需求，只要有一个新消费者进入系统时，需要进行以下操作。

向消费者发送一封欢迎邮件。

向邮局查证消费者的地址。

一种方式

这是所有的需求吗？未来还会有变化吗？

如果能相当肯定已经知道所有需求，那么将通知行为硬编码到Customer类中，就可以解决这一问题，如图18-1所示。

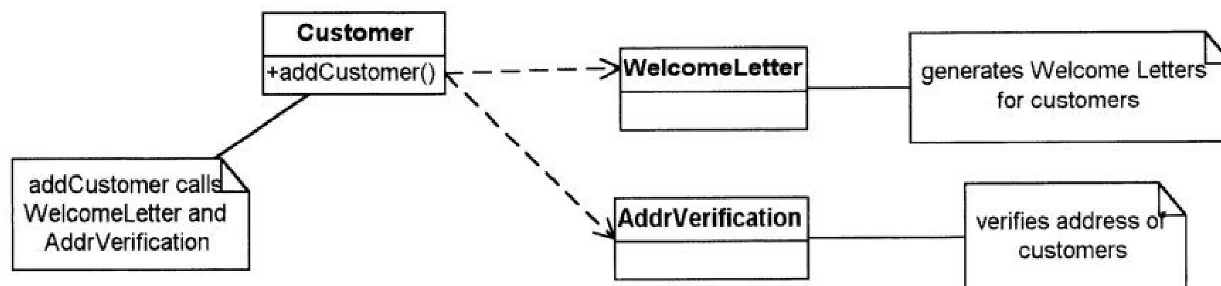


图18-1 将行为硬编码

例如，采用与在数据库中添加新消费者同样的方法，也可以调用生成欢迎邮件的对象和查证邮政地址的对象。

这些类的责任如下：

类	责　任
Customer	添加一个消费者时，该对象将调用其他对象，进行相应的操作
WelcomeLetter	为消费者创建欢迎邮件，让他们知道自己已添加到该系统中
AddrVerification	该对象将按要求查证消费者的地址

问题怎样了？需求总在变化

硬编码的方式到目前为止能够奏效。但是需求总在变化。我知道还会有另一个需求将再次要求Customer类的行为改变。例如，可能必须支持不同公司的欢迎邮件，所以需要为不同公司创建不同的Customer对象。当然，我们还有更好的办法。

18.4 Observer模式

意图，按照《设计模式》一书的说法

《设计模式》一书中对Observer模式的意图是这样叙述的："定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知并自动更新"[2]。

这意味着自动处理通知

经常会有这样一组对象，当某个事件发生时，这些对象都需要得到通知。我希望这种通知能够自动发出，但是，我不希望每次侦听通知的对象集改变时，都要修改通知对象。（这就好像每当有新的车载收音机进城时，城里的无线电台都要进行改变一样。）我希望将通知者和被通知者解耦。

一个常用的模式

这是一个很常用的模式。又称依赖（Dependents）或发布—订阅（Publish-Subscribe）[3]，与COM中的通知过程也很类似。Java中，该模式是通过Observer接口和Observable类（后面将进一步讨论）来实现

的。在基于规则的专家系统中，经常通过守护（daemon）规则来实现。

18.5 将Observer模式应用到我们的案例研究

两个东西在变化

我的方式是在问题中寻找“变化”的线索，然后，尝试封装变化。在这个案例中，我发现：

不同类型的对象——有一系列对象需要在状态发生变化时获得通知。这些对象往往属于不同的类；

不同的接口——因为它们属于不同的类，所以往往有不同的接口。

第1步：让观察者以同样的方式工作

首先，必须找出所有希望获得通知的对象。我将这些对象称为观察者（observer），因为它们在观察一个事件的发生。[4]

我希望所有的观察者对象有相同的接口。如果它们没有相同的接口，就必须修改目标（subject）——就是触发事件的对象（例如Customer对象），以处理各种不同类型的观察者。我不想这样做，因为这会使目标复杂化。

如果使所有观察者的类型都相同，目标就可以轻易地通知它们。为了使所有观察者的类型都相同：

在C#和Java中，可以用一个接口（为了灵活性或出于必要）；

在C++中，可以按需要使用单一继承或多重继承。

第2步：让观察者注册自己

在大多数情况下，我希望观察者负责了解自己观察的是什么，而且目标无需知道有哪些观察者依赖于自己，为此，观察者需要有一种方式向目标注册。因为所有的观察者都是相同类型的，所以必须在目标中添加以下两个方法：

attach(Observer)——将给定的Observer添加到目标的观察者列表

中；[5]

detach(Observer)——从目标的Observer列表中删除给定的观察者对象。

第3步：事件发生时通知观察者

既然 Subject 对象有了已注册的 Observer 对象，事件发生时，Subject对象通知Observer对象将非常简单。为此，每个Observer类都要实现一个名为update的方法。

Subject类将实现一个 notify方法来遍历其 Observer对象列表，并调用每个Observer对象的update方法，该update方法应该包含处理事件的代码。

第4步：从目标获取信息

然而，只是通知每个 Observer对象并不够。Observer对象除了知道事件发生之外，可能还需要关于该事件的更多信息。因此，必须在Subject类中添加方法，使Observer对象能够获取所需的任何信息。图18-2展示了这一解决方案。

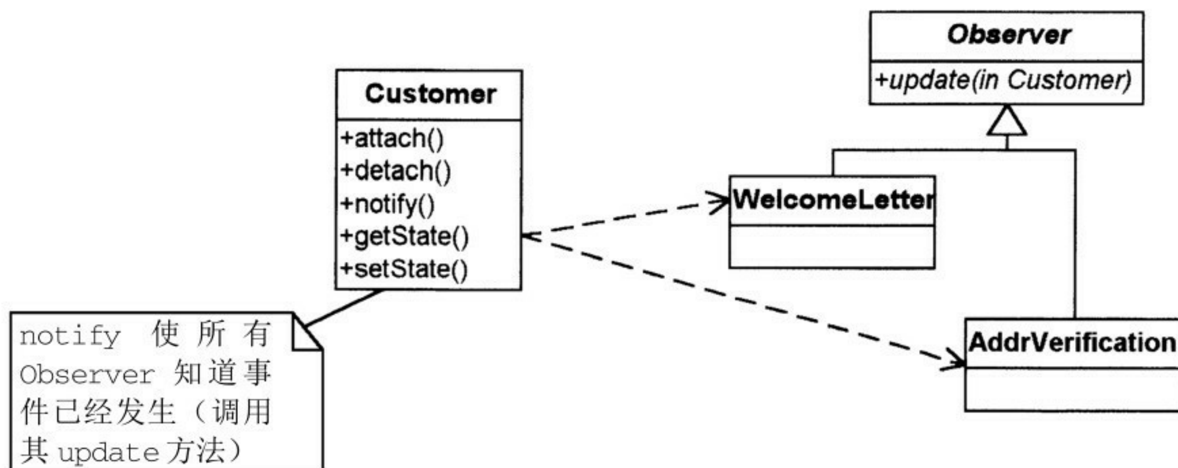


图18-2 用Observer模式实现Customer类

工作原理

在图18-2中，各个类之间的关系如下所述。

1.Observer 对象在实例化时将自己添加到 Customer 类。如果Observer对象需要从目标（Customer）那里获得更多信息，就必须传给update方法一个指向调用对象的引用。

2.当添加了一个新的 Customer 对象时，notify 方法将调用这些Observer对象。

每个 Observer对象都要调用新添加的 Customer对象的 getState方法，获取该对象的信息，明确需要执行的操作。请注意，通常会调用几个方法以获取所需的信息。

在这里，请注意我们使用了静态的 attach方法和 detach方法，因为观察者希望得到所有新Customer对象的通知。通知观察者时，将传给它们新建的Customer对象的引用。

例18-1所示为实现这一设计所需的代码。

例**18-1 Java**代码片段：实现**Observer**模式

```
// 说明：我不使用 Java Observer 或 Observable。
// 它们在实际工作中对我帮助不大
// 我不喜欢它们的接口

import java.util.*;

public class Customer {
   static private Vector myObs;
   static {
      myObs= new Vector();
   }
   public static void attach(MyObserver o){
      myObs.addElement(o);
   }
   public static void detach(MyObserver o){
      myObs.remove(o);
   }
   public String getState () {
      // 实际开发时要使用其他方法来取得必要的信息，
      // 这里返回null可通过编译
      return null;

   }

   public void notifyObs () {
      // 进行必要设置，以便观察者知道发生了什么情况
      for (Enumeration e = myObs.elements();
         e.hasMoreElements() ;) {
            ((MyObserver) e).update(this);
      }
   }
}

interface MyObserver {
   void update (Customer myCust);
}

class AddrVerification implements MyObserver {
   public AddrVerification () {
   }
   public void update ( Customer myCust) {

      // 在这里验证地址可以通过使用myCust
      // 取得相关客户的更多信息
   }
}
```

```
class WelcomeLetter implements MyObserver {
   public WelcomeLetter () {
   }
   public void update (Customer myCust) {
    // 在这里处理与欢迎信有关的操作，
    // 可以通过myCust获得有关客户的更多信息
   }
}
```

Observer模式有助于灵活性和解耦

通过这种方式我可以在不影响任何已有的类的情况下，添加新的Observer 类。它也保持了一切类之间的较松耦合。如果保证所有对象都对自己负责，这种组织方式就可以奏效。

新需求：还要发送优惠券

如果又有一个新需求，这种组织方式将表现如何呢？例如，如果我需要为距离这家公司的一个传统商店[6]20　英里之内的消费者发送一封含优惠券的信，情况又会怎样呢？

为了实现此需求，仅需添加一个新的观察者来发送优惠券，该观察者只针对那些居住在指定定距离之内的新消费者。我可以将这个观察者命名为BrickAndMortar，让它成为Customer类的观察者。这种解决方案如图18-3所示。

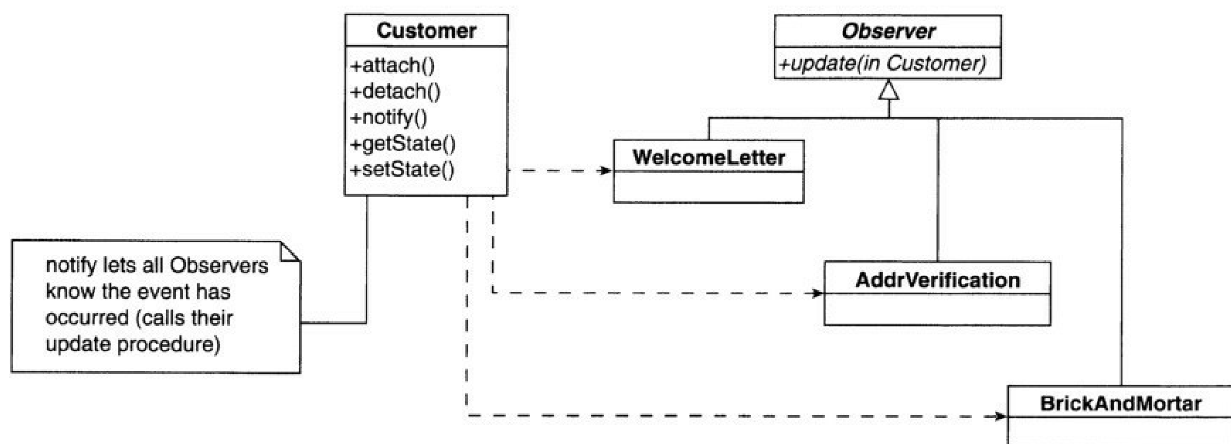


图18-3 添加BrickAndMortar观察者

实际中的Observer模式

有时候，一个要成为观察者的类可能已经存在，这时，可能不希望对其进行修改。如果这样，可以很容易地使用Adapter模式进行转换。图18-4给出了一个例子。

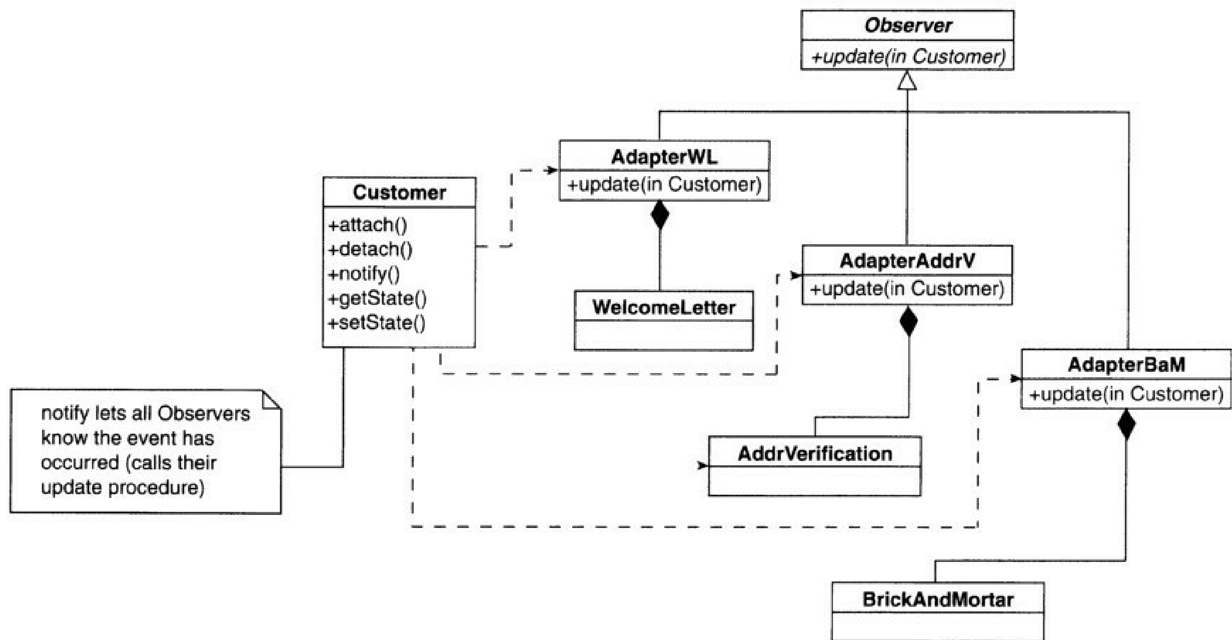


图18-4 用Adapter模式实现Observer模式

Observable类：针对**Java**开发人员的说明

Observer模式非常有用，以至于Java在其工具包中就实现了Observer模式，Observable类和Observer接口组成了这个模式，其中Observable类扮演了《设计模式》一书该模式描述中Subject的角色。Java并没有使用attach、detach和notify这些方法，而是将之分别取名为addObserver、deleteObservers和notifyObervers方法（Java使用了update方法）。Java还提供了更多方法，使我们使用起来更加方便。[7]

18.6 实践注记：使用Observer模式

并不适用于所有依赖关系

Observer模式并不是只要在对象间存在依赖关系时就要使用。例如，假设在一个票据处理系统中，有一个Tax对象处理缴税问题，显然当票据中的项目增加时，Tax对象必须得到通知以便重新计算税额。但这并不适合使用Observer模式，因为这种通知事先已经知道，而且不可能再添加其他观察者。当依赖关系固定（或实际上固定）时，引入Observer模式可能只会增加复杂性。

如果需要得到某事件通知的对象列表是变化的，或者是有条件的，那么Observer 模式更具价值。这些变化可能是由于需求的改变，或者由于需要通

……而是适用于变化或动态的依赖关系知的对象列表的变化而引起的。如果系统在不同的情况下运行，或由不同的用户运行，需要的观察者列表都会不同，这时Observer模式也很有用。

一个观察者可能只需要处理事件的某些情况。传统商店就是这样的一个例子。在这种情况下，观察者必须将额外的通知筛选掉。

是否处理某个事件

将筛选通知的责任转给 Subject 对象，可以避免额外的通知。最好的实现方式是 Subject 对象使用一个 Strategy 模式测试通知是否应该发出。每个观察者在注册时都将正确的Strategy对象提供给Subject对象。

怎样处理某个事件

有时候，Subject会调用观察者的update方法，传递信息。这可以避免观察者回调Subject，但是，经常是不同的观察者有不同的信息需求。在这种情况下，可以再次使用Strategy模式。这时，用Strategy对象调用观察者的 update方法。同样，观察者必须将正确的Strategy对象提供给

Subject对象使用。

Observer模式：关键特征

意图

在对象之间定义一种一对多的依赖关系，这样当一个对象的状态改变时，所有依赖者都将得到通知并自动更新。

问题

当某个事件发生时，需要向一系列变化着的对象发出通知。

解决方案

Observer将监视某个事件的责任委托给中心对象：Subject。

参与者与协作者

Subject知道自己的 Observer，因为 Observer要向它注册。Subject必须在所监视的事件发生时通知Observer。Observer负责向Subject注册，以及在得到通知时从Subject处获取信息。

效果

如果某些Observer只对事件的一个子集感兴趣，那么Subject可能会告诉它们不需要知道的事件（参见18.6节）。如果Subject通知Observer，Observer还返回请求更多信息，则可能需要额外的通信。

实现

让某个事件发生时需要知道的对象（Observer）将自己注册到另一个监视事件发生或自己触发事件的对象（Subject）上。

事件发生时，Subject告诉Observer事件已经发生。

为了对所有Observer类型的对象实现Observer接口，有时候需要使用Adapter模式。

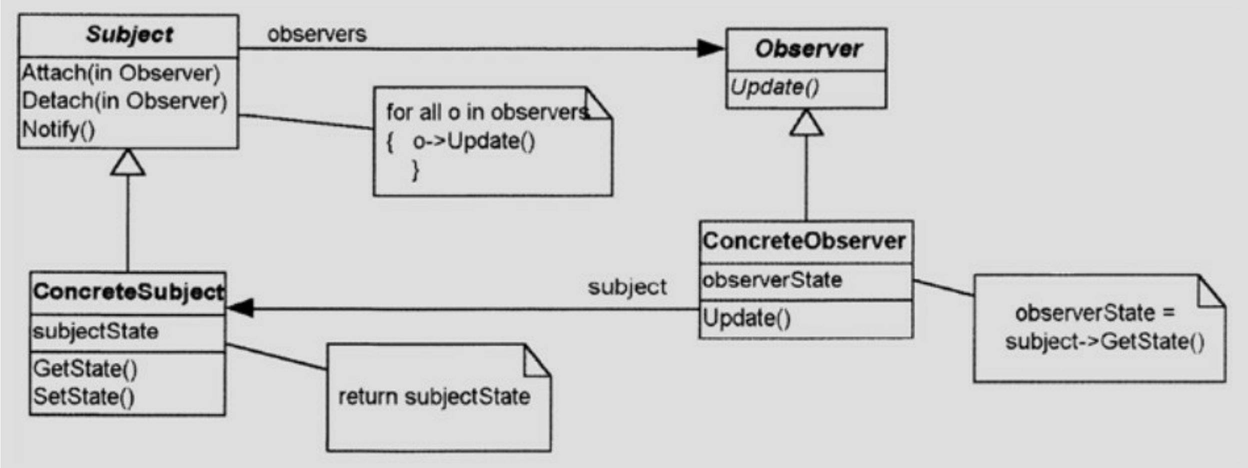


图18-5 Observer模式的通用结构图

18.7 小结

本章内容

学习Observer模式过程中，我观察了哪个对象能最好地应对未来的变化。对于Observer模式，触发事件的对象——Subject对象无法预测可能需要知道该事件的所有对象。为了解决这一问题，我创建了一个Observer接口，要求所有的Observer负责将自己注册到Subject上。

所用面向对象原则的总结

这一章所讨论的是 Observer 模式，但 Observer 模式用到的几条面向对象原则也非常值得重点指出。

概　　念	讨　　论
对象自我负责	Observer 对象有多种，但都从 Subject 对象收集所需信息，并自己完成相应的操作
抽象类	Observer 类表示了“需要得到通知的对象”这一概念。它为目标提供了一个通知 Observer 的公共接口
多态封装	Subject 不知道与哪种观察者通信。实际上，Observer 类封装了各种特定的 Observer。这意味着如果我在未来有新的 Observer，不需要修改 Subject 类

复习题

简答题

1.按照《设计模式》一书，结构型模式负责的是什么？

2.按照《设计模式》一书，有哪三种模式分类？作者建议的第四种模式是什么？

3.关于需求的一个事实是什么？

4.Observer模式的意图是什么？

阐述题

1.为什么Bridge和Decorator模式归为结构型模式比归于行为型模式更正确？

2.Observer 模式的一个软件之外的例子是无线电台。它对外广播信号；任何感兴趣的人都可以调整频率，收听自己想听的节目。你能从“真实生活”中再举出一个例子吗？

3.在什么情况下不应该使用Observer模式？

观点与应用题

模式的“第四个分类”含有其他分类中的模式，这样分类好吗？给出你的理由。

第19章 Template Method模式

19.1 概览

本章内容

本章将继续讨论第9章、第16章和第18章已经讨论过的电子商务案例研究。

在本章中，我们将：

通过讨论这个案例中增加的需求介绍Template Method（模板方法）模式；

描述Template Method模式的意图；

给出Template Method模式的关键特征；

描述Template Method模式如何消除冗余；

描述我在实践中使用Template Method模式的一些经验。

19.2 案例研究的更多需求

新需求：访问多种SQL数据库系统

在编写国际电子商务应用程序的过程中，假设我收到一个新需求：支持Oracle和SQL Server数据库。这两个系统都基于能够使数据库使用更加简单的通用标准SQL（结构化查询语言），但是，虽然在一般层次上SQL是一个通用标准，但是两个系统的细节仍然存在差别。

在一般层次上，步骤是相同的……

例如，我知道一般而言，查询这些数据库时，需要采取以下步骤。

1.格式化CONNECT命令。

2.将CONNECT命令发送给数据库。

3.格式化SELECT命令。

4.将SELECT命令发送给数据库。

5.返回选中的数据集。

……但细节不同

但是数据库的具体实现不同，所需的格式化过程也稍有差异。

19.3 Template Method模式

对步骤标准化

Template Method 是一个旨在帮助我们在抽象层次从一组不同的步骤中概括出一个通用过程的模式。按照《设计模式》一书的说法，Template Method模式的意图是：

定义一个操作中算法的骨架，而将一些步骤延迟到子类中。不改变算法的结构而重定义它的步骤。[8]

也就是说，尽管连接和查询Oracle数据库或SQL Server数据库有不同方法，但它们的过程在概念上是相同的。Template Method 模式给我们提供了一种方式，它能够在一个抽象类中捕捉共同点，同时在派生类中封装差异。Template Method模式其实就是控制不同过程中共同的序列。

19.4 将Template Method模式应用到我们的案例研究

细节是变化的

在国际电子商务案例中，数据库访问中的变化出现在相关步骤的特定实现中，如图19-1所示。

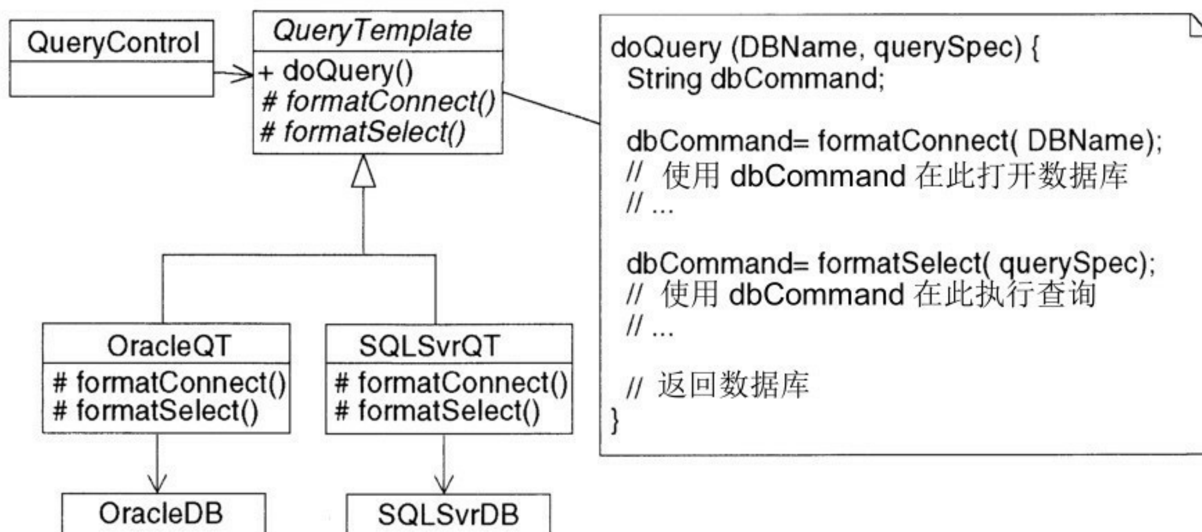


图19-1 使用Template Method模式执行查询

工作原理：为变化的步骤提供虚方法

我创建了一个名为 doQuery 的方法，来处理需要执行的查询。将数据库名和查询规范传给它。doQuery方法完成上面的五个一般步骤，为每个步骤都提供虚方法（如 formatConnect和formatSelect等），这些虚方法的实现都不相同。

doQuery方法的实现如下：如图19-1所示，首先需要格式化连接数据库所需的 CONNECT 命令。虽然抽象类（QueryTemplate）知道需要格式化，但它不知道如何完成。真正的格式化代码是由派生类提供的。格式化SELECT命令也是如此。

Template Method 模式之所以如此要求，是因为方法是通过一个指向某个派生类的引用来调用的。也就是说，虽然QueryControl对象有一个类型为 QueryTemplate的引用，但它实际之下的是一个 OracleQT或者SQLSvrQT对象。因此，调用这些对象的doQuery方法时，所确定的方法将首先寻找合适的派生类的方法。假设我们的QueryControl对象使用的是一个OracleQT对象。因为OracleQT类没有覆盖QueryTemplate，所以将调用 QueryTemplate 的 doQuery 方法。它开始运行，直到调用

formatConnect方法。因为请求的是OracleQT对象执行doQuery方法，所以将调用 OracleQT 类的 formatConnect 方法。然后，控制又返回QueryTemplate类的 doQuery方法。现在执行所有查询公共的代码，直到需要下一个变化——formatSelect 方法。同样，这个方法位于QueryControl对象引用的对象（本例中是OracleQT）。

当遇到新数据库时，Template Method 模式提供了一个可以填充的模板。我们创建一个新派生类，实现新数据库所需的具体步骤即可。

19.5 使用Template Method模式减少冗余

消除冗余会带来抽象

我的咨询实践中，曾经多次与非常聪明但是面向对象背景不强的团队合作。他们也在转向敏捷方法，而且懂得消除冗余、强内聚和松耦合等概念的必要性，但是，他们不知道如何实现这些要求。

有一次，我请一个小组的负责人描述一下他们遇到的典型问题。他对我说，他们不得不支持几个不同合作伙伴公司的系统。规则大体相同，但总是存在一些微妙的差异。因为到处都散布着许多检查当前状态和如何进行处理的if-then-else语句，代码变得越来越难以维护。他只想到了以下两个方案，但是都不好：

继续加入更多的if-then-else语句，这将进一步使代码变坏；

为每一种情况复制和粘贴代码，从而导致重复。

使用if-then-else语句或者分支语句是经常采用的方式。问题在于它会导致所谓的分支蔓延。开始的时候，少量的if-then-else语句或者分支语句还不是什么大问题，但是总会有一天代码将变得难以理解。复制粘贴方式的好处是虽然不那么干净利落，至少每一部分是清楚地，因为只与一种情况相关。

因为懂得Template Method模板，所以我能够提出第三种方案。我将

分两步说明这一方案。首先要说明人们复制和粘贴后，如果代码需要更新时会发生的情况。事实证明即使所有代码都修改了，仍然存在公共的过程，因为如果不是这样就没有理由复制粘贴了。其次，我将说明在重新组织了这一重复过程之后，如何重构代码，消除重复。

复制粘贴然后修改代码会留下冗余

图19-2所示为用来说明复制粘贴缺点的一段“代码”示例。请注意，给出的所谓“代码”只是一些字母序列，因为代码的细节并不重要。这是为了更容易地看到复制粘贴的问题。

```
MyClass

someMethod () {
  aaa aaa aaa aaa
  bb bb bb bb bb
  cccc cccc cccc
  d d d d d d d d d
  eee eee eee eee
  fffff ff fffff ff
  ggg gggg ggg
  h hh h hh h hh h
  iiiii iiiii iiiii iiiii
}
```

图19-2 原代码

使用复制粘贴修改代码之后，最后可能得到一个新的代码序列，如图19-3所示。请注意在这种复制粘贴的操作中，有些a已经转变成了A，而有些b变成了B，等等。字母变为大写时，表示代码是复制粘贴然后修

改的。如果仍然为小写，则说明代码只是简单复制。新的一行X是为新情况添加的代码，在原代码中不存在。

不同种类的重复

这里至少存在两类重复。第一类更加明显一些：c、f和I行是重复的代码。我复制粘贴了原代码，然后修改了大部分，但不是全部。没有修改的部分显然是冗余的。但是，还有另一类重复。代码片段有相同的操作序列。也就是说，这种复制粘贴主要用于有定义明确的步骤序列，但有些步骤的实现发生了变化的时候，如图19-4所示。

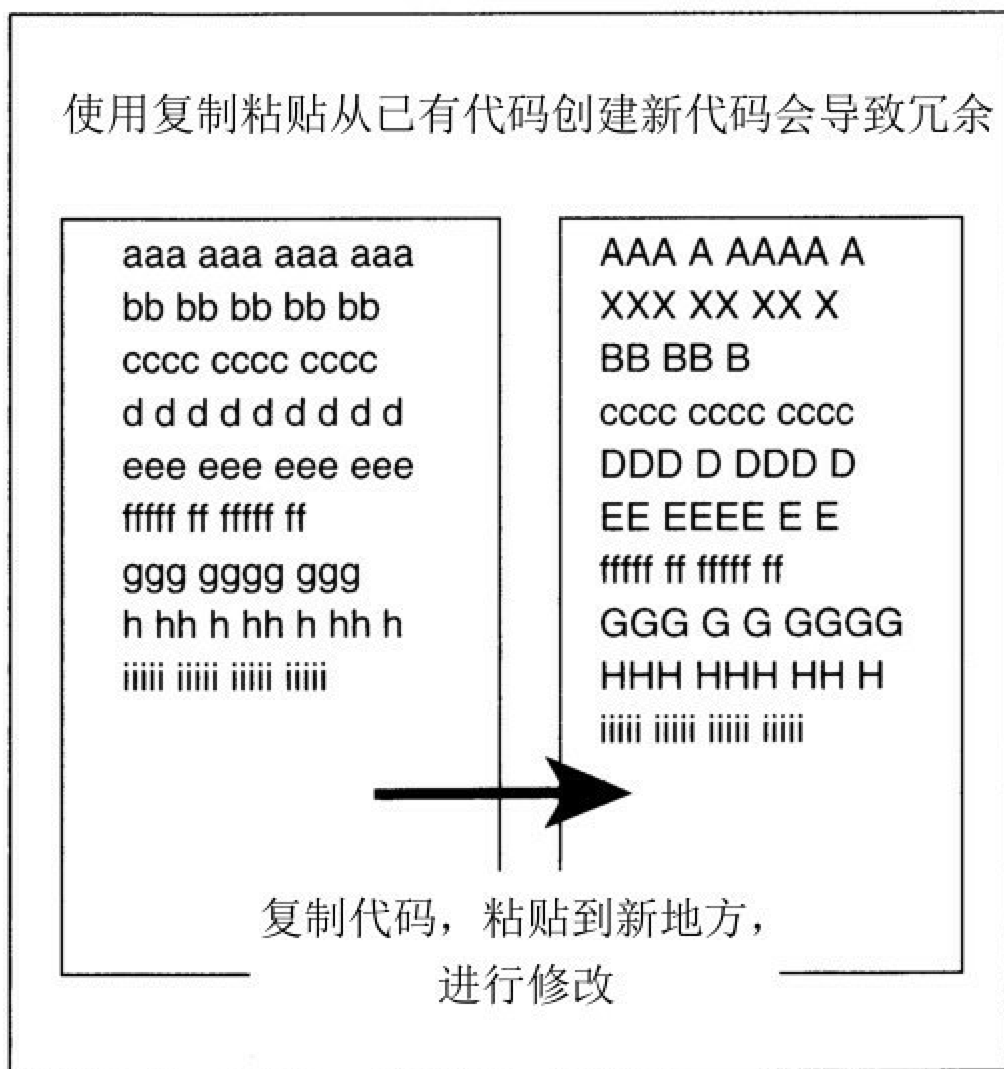


图19-3 原代码与新代码

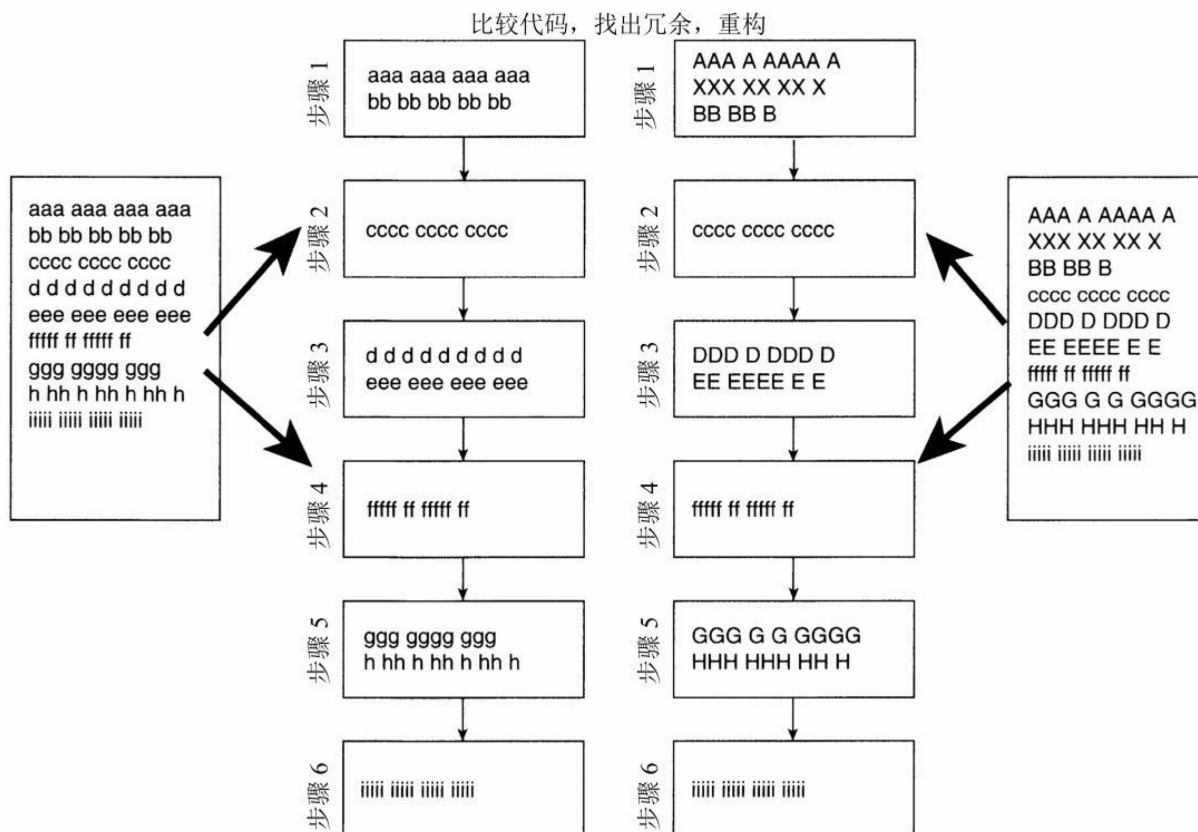


图19-4 比较代码找出冗余

这些步骤指出了概念上相同但是实现方式不同的地方（这里用大小写以示区别，有些地方是增加新代码的，用X表示）。例如，在原来的类中的步骤1由小写的a和b组成，而在新的类（第二个类）中，步骤1由大写A、X和B组成。两个类中都有该步骤，所以存在冗余，虽然实现不同。

去除重复

这里可以用Template Method模式来去除重复。它指示我们用一个基类实现步骤序列。然后对每种情况都用自己的派生类实现特殊步骤，如图19-5所示。

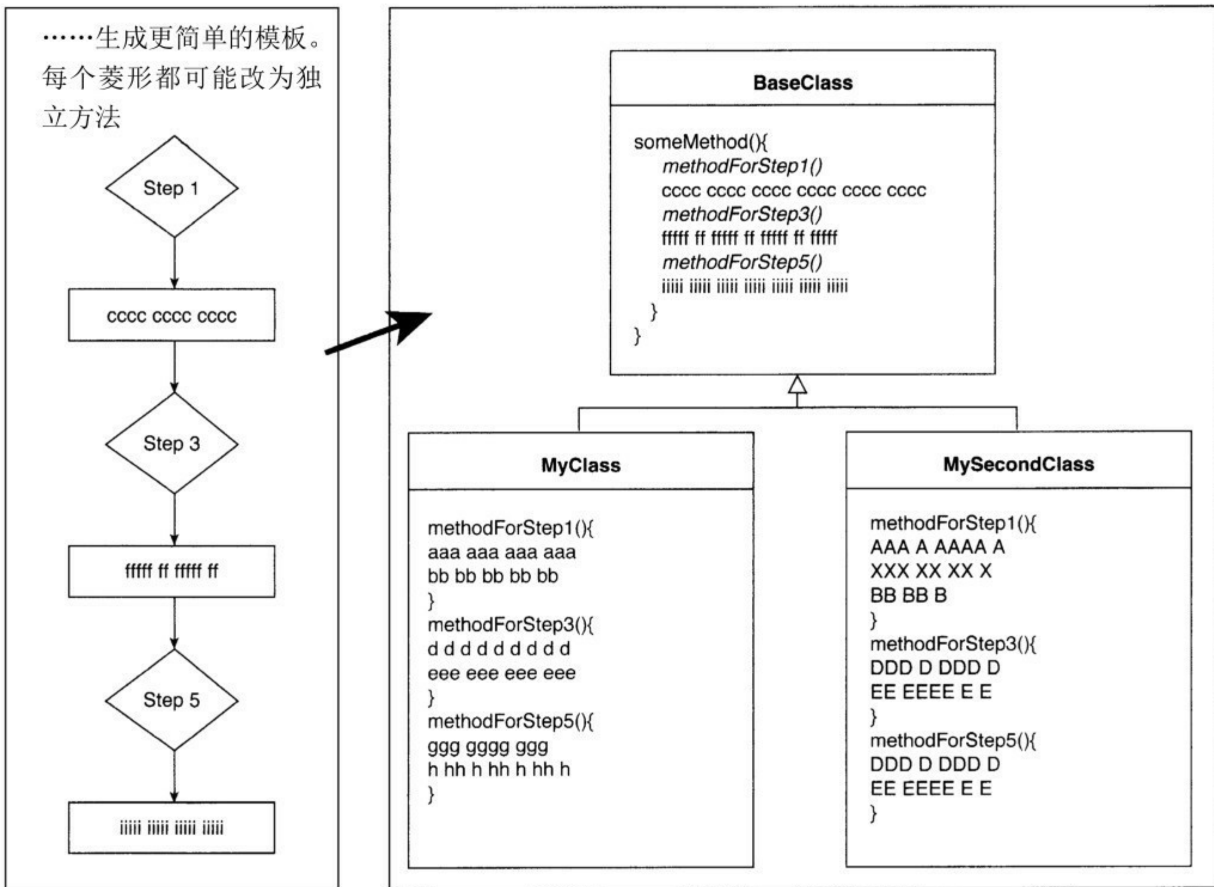


图19-5 通向Template Method模式

一些缺失的细节

要真正地实现这一点，需要填充一些细节。首先，派生类中的步骤需要在基类中声明为抽象和/或保护方法。其次，在步骤之间需要使用局部方法变量。这些变量有的需要作为参数传入，有的作为返回值，也有的是类的数据成员。各步骤公共的变量应该放在基类中。

一开始就避免重复的一种方式

当然，在发现糟糕情况之后再重构修补比不管不顾要好。但是，一开始就重构，避免出现这种糟糕情况更好。这种情况下，当需求要求第二个系统时，如果认识到Template Method 模式方式可行，可以采取以下步骤。

1.首先，重构第一个方案（使用 Extract方法重构），提取要修改的

代码，如图19-6所示。顺便说一下，“提取要修改的代码”与《设计模式》一书中的建议“找到变化并封装之”听上去有些类似。这里我没有特化方法，只用一个someMethod方法包含了要变化的其他方法。

MyClass

```
someMethod () {
   methodForStep1()

   cccc cccc cccc

   methodForStep3()

   fffff ff fffff ff

   methodForStep5()

   iiiii iiiii iiiii iiiii
}

methodForStep1 () {
   aaa aaa aaa aaa
   bb bb bb bb bb
}

methodForStep3 () {
   d d d d d d d d d
   eee eee eee eee
}

methodForStep5 () {
   ggg gggg ggg
   h hh h hh h hh h
}
```

图19-6 重构要变化的步骤

2.其次，创建一个基类包含不变的方法（someMethod）。代码现在如图 19-5 中的 BaseClass 和 MyClass 所示。（我还没有编写 MySecondClass。）

3.编写 MySecondClass。请注意现在添加新功能除了工厂类（也只是可能）之外，将不会影响任何其他代码。

当然，可以注意到在所有表明应该“提取方法”的地方，如果一开始就按意图编程，重构本来是能够避免的。这种好的实践会不断地显现出其价值，这也是它们之所以优秀的原因。

懂得模式有助于引导重构

一般而言，这是重构的一种很好的方式。当有新需求需要处理时，应该采取以下两个步骤。

1.先重构代码，不添加任何功能，这样加入新功能时可以遵守开闭原则。

2.添加新的代码，只影响工厂类和新代码。

懂得模式有助于在许多不同的情况下遵循这一方式。

19.6 实践注记：使用Template Method模式

Template Method模式并不是耦合的多个Strategy模式

有时候会有一个类使用几个不同的 Strategy 模式。第一次看到 Template Method模式的类图时，我想：“噢，Template Method模式只是一组共同协作的 Strategy 模式的集合而已。”这是一种很危险的（而且通常不正确的）想法。虽然几个Strategy模式看起来互相连接的情况并不是非常罕见，但是这种设计会影响灵活性。

Template Method模式适用于存在几个互不相同但概念上相似的过程。每个过程的变化是互相耦合的，因为它们都与某个过程相关。在前

面给出的例子中，需要格式化Oracle数据库的 CONNECT命令时，如果需要格式化Query命令，它也是针对Oracle数据库的。

懂得模式有助于提供代码质量而且不会增加工作量

我刚才讲述的故事，说明了懂得模式有助于遵循极限编程（eXtreme Programming，XP）的实践。熟练的XP设计师在这种情况下往往会想到（并在合适时实现）类似于Template Method模式的方案。如果这是避免冗余的最佳方式，就应该实现该模式。XP 设计师可能不认为自己是在“遵循模式”，但是对于模式的了解能够提供一种如果不了解模式自己可能不会想到的选择。糟糕的是，没有丰富XP经验的设计师很可能认识不到存在这样的方案。他甚至没有认识到，复制粘贴会造成冗余的过程，即使修改了所有代码。这再次确证了我的观点，即模式是一种指导。它们在XP风格的编程实践中以及在预先设计式的实践中都可以非常有效地得到应用。

TempIate Method模式：关键特征

意图

定义一个操作中算法的骨架，将一些步骤推迟到子类中实现。可以不改变算法的结构而重定义该算法的步骤。

问题

要完成在某一细节层次一致的一个过程或一系列步骤，但其个别步骤在更详细的层次上的实现可能不同。

解决方案

允许定义可变的子步骤，同时保持基本过程一致。

参与者与协作者

Template Method 模式由一个抽象类组成，这个抽象类定义了需要覆盖的基本TemplateMethod方法（见图19-7）。每个从这个抽象类派生的具体类将为此模板实现新方法。

效果

模板提供了一个很好的代码复用平台。它还有助于确保所需步骤的实现。它将每个Concrete类的覆盖步骤绑定起来，因此只有在这些变化总是并且只能一起发生时，才应该使用Template Method模式。

实现

创建一个抽象类，用抽象方法实现一个过程。这些抽象方法必须在子类中实现，以执行过程的每个步骤。如果这些步骤是独立变化的，那么每个步骤都可以用Strategy模式来实现。

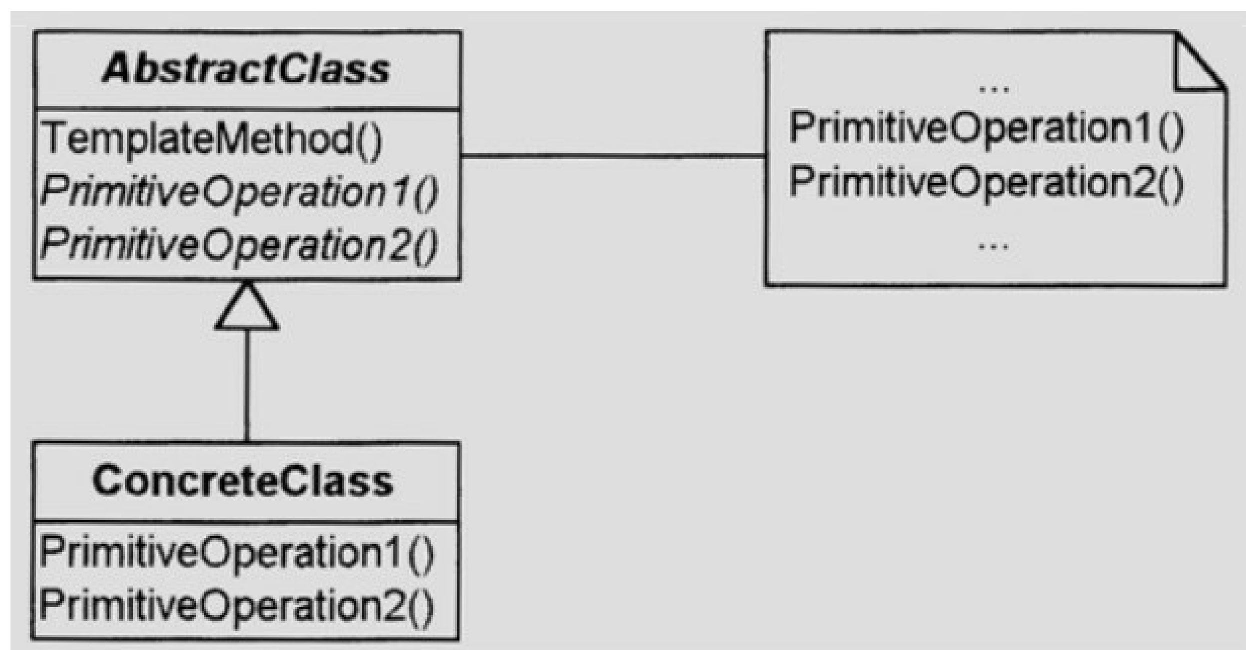


图19-7 Template Method模式通用结构图

19.7 小结

本章内容

有时候，会遇到由一系列步骤构成的过程需要执行。这个过程从高

层次上看是相同的，但有些步骤的实现可能不同。例如，查询 SQL 数据库从高层次上看过程是相同的，但某些细节比如如何连接数据库则可能因平台等细节的不同而不同。

通过Template Method模式，我们可以先定义步骤序列，然后覆盖那些需要改变的步骤。

通过Template Method模式，在需要支持概念上行为相同而每个步骤的实现不同的多个系统时，我们还可以避免冗余的出现。

复习题

简答题

Template Method 模式以一种特殊的方式进行方法调用，这种方式是怎样的？

阐述题

1.按照《设计模式》一书，Template Method模式的意图是“定义一个操作中算法的骨架，而将一些步骤延迟到子类中。在不改变算法的结构的前提下重定义它的步骤。”这是什么意思？

2.《设计模式》一书称此模式为“Template Method（模板方法）”，为什么呢？

3.Strategy模式（第9章）和Template Method模式有何区别？

---

[1].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.10。

[2].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.293。

[3].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，293页。

[4].实际上，我这么称呼它们，是因为 Observer 模式对于观察事件发生的对象就是这样命名的。这里只是为了指出这样命名的原因。

[5].这里的添加到列表就是所谓注册。——译者注

[6].原文为“brick and mortar”store，是网络泡沫时期对非网络型传统商业和服务形式的一种称谓。——译者注

[7].关于Observer接口和Observable类的Java API的更多信息，请参见http：//java.sun.com/j2se/1.3/docs/api/index.html。

[8].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.325。

第七部分 各种工厂模式

概览

本部分内容

工厂模式可以用于帮助对象的创建。但是这可能并非它们最重要的作用。换一个新角度，会发现更多。

章 讨论的主题

20 来自设计模式的教益：各种工厂模式

叙述了如果要遵循模式的原则和实践，就暗含着必须将对象的使用与对象的构造和管理分离开来，各种工厂模式应运而生。

21 Singleton模式和**Double-Checked Locking**模式

叙述了两个简单的工厂模式，它们能够确保一个类型只存在一个对象。

22 Object Pool模式

叙述了我过去曾经从事的一个项目，在这个项目中我使用从模式学到的实践推出了Object-Pool模式。这个例子还说明了模式知识怎样促进敏捷实践。

23 Factory Method模式

这个模式可以用于协调两个相关的对象层次。

24 工厂模式的总结

总结本部分中介绍的各种对象工厂/管理者的使用。

第20章 来自设计模式的教益：各种工厂模式

20.1 概览

本章内容

本章讨论工厂（创建其他对象的对象）的使用。虽然《设计模式》一书中介绍了工厂，但是它们在设计中的使用书中并没有完整地予以讨论。

在本章中，我们将：

讨论为什么使用工厂会极大地简化设计和代码；

阐述正确的步骤：先了解对象，然后再决定如何创建和/或管理它们。

20.2 工厂

工厂封装了创建对象的业务规则

通常，当我设计对象时，关心的是它们的行为：对象做些什么，其他对象怎样告诉他们做这些。但是我还需要关心何时需要特定的对象，以及确保需要时它已经准备就绪。我需要确保对象创建的规则得到遵守。

新接触面向对象编程的开发人员经常不能很好地处理对象创建和对象实例化。我相信这是因为使用过程语言时，开发人员对手边的具体情况了如指掌。他们是在实现层次进行思考。而面向对象语言向他们提出了新问题，也提供了新机会。在面向对象编程中，对象在各种不同时候出于不同原因而产生。如果使用对象的代码同时需要负责实例化该对象

（经常如此），代码将变得非常复杂。它必须了解许多事情：要创建哪些对象，需要哪些构造参数，在构造之后如何使用对象——甚至经常还包括如何管理一个对象池。这会降低内聚性，这种后果可不是我们所愿意看到的。它还经常会使开发人员在对需要实例化的对象有充分了解之前过早选择实例化方式。我更愿意在了解到最适合的使用方式之前，不固定一种方式。

工厂能够解决这些问题。它们有助于保持对象内聚、解耦和可测试；它们有助于保持设计的灵活性；它们使我可以将问题分割成多个更小、更易处理的部分。

何谓工厂？

但是首先，我们来定义“工厂”的含义。工厂是用来实例化其他对象的方法（静态或者非静态）、对象或者其他任何实体。在第11章中我们已经给出了一个工厂的例子。这是一种极为复杂的工厂，因为它不是仅实例化一个对象，而是控制着一组（一系列）对象。《设计模式》一书描述了几个与工厂有关的模式，书中称它们为创建模式，包括如下几个：

Abstract Factory模式；

Builder模式；

Factory Method模式；

Prototype模式；

Singleton模式。

《设计模式》一书根据一般目的定义模式

《设计模式》一书的一般分类建立在模式的概念性动机上：

行为型模式用于封装行为的变化；

结构型模式用于将已有的代码集成到新的面向对象设计中；

创建型模式[1]用于管理对象的创建。

在我们以新方式处理变化的背景下，这非常重要。定义新对象时，

我按自己的需要定义它们，使用行为型模式作为指导。但是，当我要将已有的对象加入新的设计中时，使用结构型模式作为指导最为合适。

而使用工厂是隐藏变化的一种自然结果。考虑一下Strategy模式，如果模式中的Context对象（第9章例子中的SalesOrder类）不知道使用的是哪个Tax对象，那谁知道呢？最简单的回答是有其他对象知道。这个“其他对象”可能就是一个工厂。

20.3 再谈背景

工厂大背景

在第13章中，我们讨论了“软件开发的大背景”。

规则：先考虑系统中需要什么，然后再去关注如何创建系统……也就是说，我们应该在确定了对象是什么之后再定义工厂。

首先，确定对象是什么，它们如何工作，然后确定如何对其实例化。为了说明这样做的重要性，我们来做一个小的脑力测试。

想象我们曾经开发的一个项目中有许多不同的情况需要处理——大量变化、if语句和switch语句。在编写所有代码之前，假设团队负责人说：“我们将分成两组，A组将决定我们的对象是什么，还有它们如何协作。B组的任务是在各种情况下实例化正确的对象。”

也就是说，A组将根据本书中概述的规则设计对象，仅仅设计它们的工作方式。这样客户对象就无需操心所有不同的变化了。事实上，B组将为涉及到的每种情况编写负责实例化正确对象的类和对象。B组的代码实际上不仅仅实例化对象，有时候对象还需要共享和复用。所以，它们的“工厂”实际上是“负责管理的工厂”，但是我将仍然称之为工厂。

现在我们来考虑两个问题。首先，哪一组任务更艰巨？是定义对象并确定如何使用的一组，还是只知道需要哪些对象、如何创建和管理这些对象的一组？我提问的所有学员几乎读认为B组的任务更轻松。他们

有许多相当明了的规则，而且彼此基本上是解耦的。根据实际情况实现何时创建对象的规则，相对要容易一些。

接下来，考虑这个问题：B 组的工作在多大程度上简化了 A 组的工作？对于通常的项目，我的回答都是“非常大”。原因在于，复杂代码之所以一般比较困难，是因为它必须处理面临的特定情况，并确定此情况下使用哪些特定的对象。采取这种方式后，A组只需按接口和抽象类设计，B组要弄清实际使用的是哪些实现和派生。两个组的工作都变得容易了。简化对象的创建和管理之后，我们就能够事半功倍。

为给定情况下所用的对象组织规则，是一个相对简单的问题，将极大地简化使用对象的代码。我推荐这种方式，如图20-1所示。

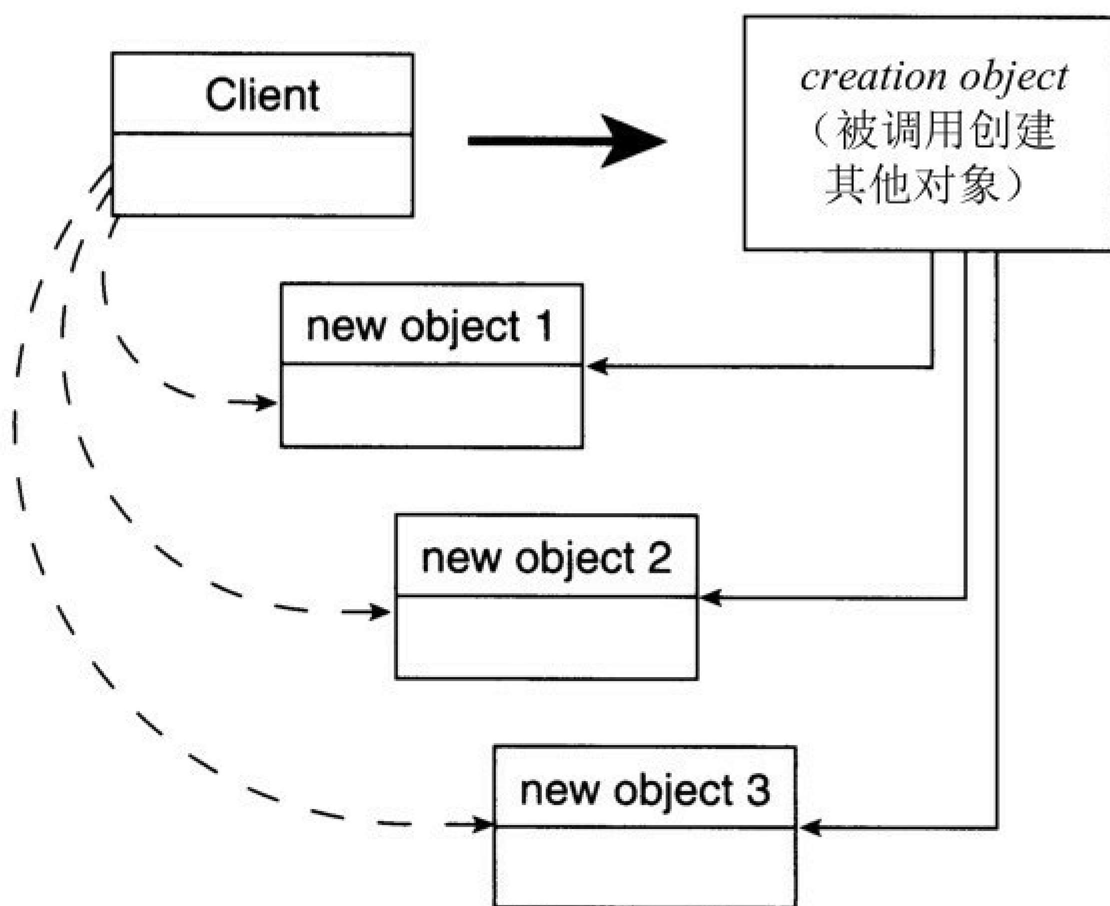


图20-1 客户使用工厂构造对象然后使用对象

20.4 工厂遵循我们的准则

工厂能够提高内聚性，松散耦合，并有助于测试......

考虑将开发分为以下两步的方法。

1.定义对象和它们的协作方式。

2.编写为相应情况实例化对象并在对象共享时管理已有对象的工厂。

步骤1中生成的代码无需操心哪个对象应该实例化，而步骤2中的代码则无需操心对象的协作方式。也就是说，两个步骤代码的内聚性都更好。为什么呢？如果不遵循这样的方式，代码就必须既处理功能，又负责决定在不同情况下应该构造和/或管理哪个对象的规则。将这些问题分离，在某一个类中就只需处理其中之一。

规则：对象要么构造其他对象，要么使用其他对象，决不要两者兼顾

对于对象的创建和管理，有一条很好的通用规则可以遵守：对象应该要么构造和/或管理其他对象，要么使用对象，而不应该兼而有之。

如果遵守这一约束，最终将降低耦合度，因为分工非常明确。“使用对象”与它们使用的对象是解耦的。它们无需知道有什么协作对象，甚至不需要知道可能会有什么对象，这些工作是工厂负责的。同时，工厂只知道它们在创建或者管理哪些对象，而无需知道这些对象如何使用。这种划分如图20-2和图20-3所示。

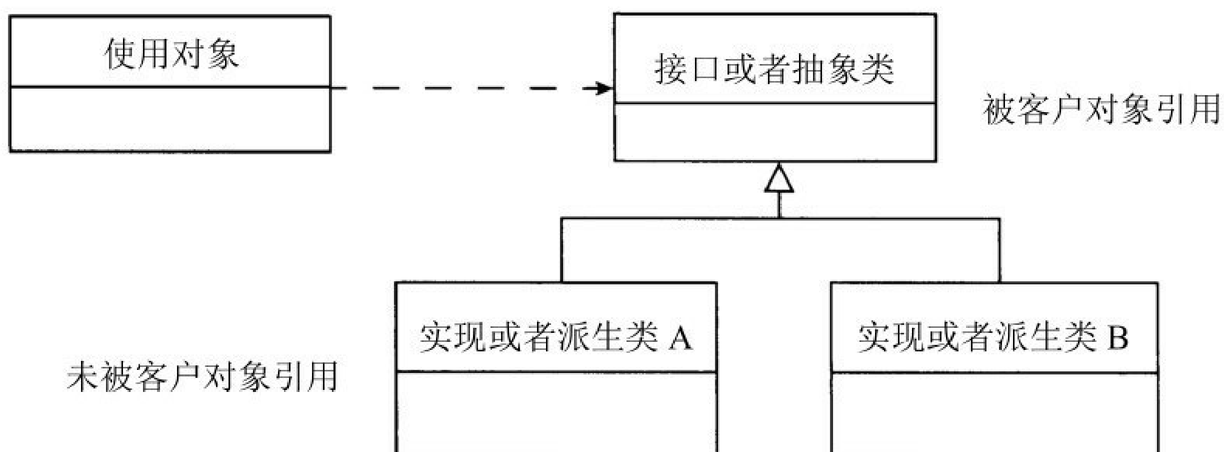


图20-2 “使用对象”的视角

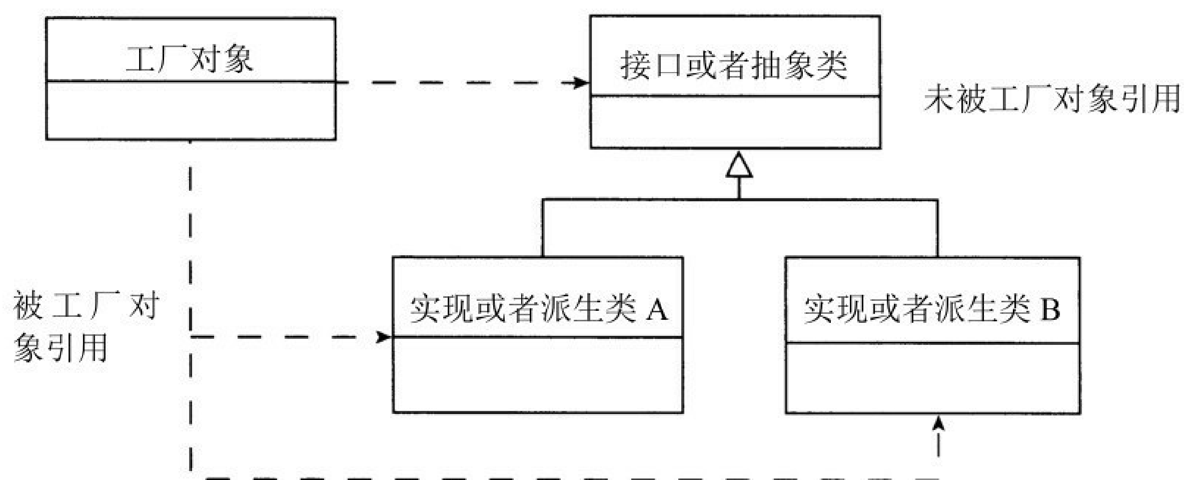


图20-3 “创建对象”的视角

请看图20-2。“使用对象”[2]只知道如何使用所引用的对象，不知道有什么具体的对象。而另一方面，图 20-3 中可以看到“工厂对象”知道有哪些对象，但是不知道如何使用它们。这种关注点的分离既加强了内聚性，又降低了耦合度。

而且，这种方式还为封装创造了更多机会，之前我们说过这是非常重要的一点。实现类对“使用对象”完全隐藏了。添加新的实现或者删除一些已有的实现，都不会改变“使用对象”。

这还有助于测试，因为“使用对象”的行为方式应该与任何派生类对象相同。这样也就无需测试所有可能的组合，因为我可以单独测试各个

部分。无论如何组合，系统的工作方式是一致的。

20.5 限制变化的影响

模式有助于我们关注的主要问题之一，就是使未来的维护更容易，成本更低。一般而言，模式有助于我们遵循开闭原则，即需要修改时，应该添加新代码，而不是修改老代码。这能够降低维护成本。

这里工厂发挥了主要作用。

虽然一个对象经常有许多“使用者”，但是通常只有一个工厂负责构建和/或管理它。所以，代码的修改越能够限于工厂内，则主系统上的工作就越少——修改主系统通常是成本最高的。

需要修改时，通常要么是修改需要的对象，要么是改变使用某些对象的方式。工厂有助于将维护限制在对象的使用者或者构建者上，同时修改二者的机会将大大降低。

知道了可以“以后再操心实例化”，我们就能够关注于设计中“可插拔的”那些方面。这使我们解放出来，更可能看到灵活的方案。

20.6 对工厂的另一种思考方式

使集成更加容易

在第15章中讨论共性与可变性分析的时候，我提到过通常在一个系统中集成新代码，比从头开发的成本还要高。将使用与构造分离将降低集成成本。这是因为只需要处理新的情况，而且添加新情况的框架已经有了。

遇到难以理解和/或集成的代码时，经常会发现代码的主要部分都是必须不断地跟踪目前所处的特定情况。有了工厂处理这些之后，就没有这种问题了。因为本质上，所有与情况有关的信息都是放在几个对象中。这样弄清各种情况之间的关系就容易多了。换言之，工厂并不能完

全消除工作量，但是它能够在必须处理新情况时，避免使已经很复杂的代码更加复杂。

工厂使我们能够充分利用面向对象方法的优点，而不是仍然使用大量的过程逻辑。我们不用再用switch语句根据某种行为的变化而分支执行，只需改变起作用的对象即可。通过工厂，使用和控制行为都更加容易。

20.7 工厂的不同角色

不仅包括哪些对象

第16章说明了如何正确使用Abstract Factory模式使SalesOrder只需处理税、运费等等。添加新的实现经常只涉及创建新情况并更新工厂。但是，我们也提到，工厂还能管理对象。有时我们希望共享对象，只用一个或者有限数量的对象，数据库连接就属于此类。所以工厂应该既关心使用哪些对象的规则，又需关心如何管理它们的规则，可能包括实例化多少对象，如何共享对象。这使我们能够在工厂中封装所用对象的所有规则。

工厂能封装设计

第17章中，我们见到了对象链如何置于代码认为自己正在使用的对象之前。在这个模式和其他《设计模式》一书中提到的模式里，对象的构造和将对象插入设计中可能是非常复杂的。工厂能够隐藏这种复杂性，使“使用对象”相信自己只是在处理一个简单的对象，从而实现了设计的封装。

有读者可能会说工厂所解决的正是模式带来的问题。模式的方式是在代码中增加间接层次，因为它要获得维护性更好的对象。但是，这也加重了客户代码的负担，迫使它知道所有这些对象，从而使代码更加复杂。工厂隐藏了这些额外对象存在的事实，这简化了代码。

20.8 实践注记

创建对象中的常见问题

在尝试遵循本章中介绍的准则时，需要明白类似的重现的问题，会出现在不同情况下。对许多问题都会得到与设计模式完全相同的具体解决方案。你已经见到了其中之一——Abstract Factory。在下面的章节我们将讨论其他一些。

从这里再进一步，复杂的创建问题经常可以通过将行为型和结构型模式结合到工厂逻辑中得到解决。例如，我曾经创建过使用Bridge模式的工厂和使用 Decorator 模式的工厂。我还创建过用到本书中没有介绍到的模式的工厂，其中包括Chain of Responsibility（责任链）模式，该模式确定应该实例化哪个对象来处理一个给定的协议对象。

20.9 小结

本章内容

本章阐述了为什么工厂能够使简化我们的工作。工厂封装了在什么环境下创建什么对象的规则。这样当系统的其他部分使用对象时，就可以不考虑具体的实现。

复习题

简答题

1.给出“工厂”的定义。

2.举出一个前面章节讲述过的工厂模式。举出一个本章提到的模式。

3.管理对象创建时用到的一个好的通用规则是什么？

阐述题

1.新接触面向对象编程的开发人员经常会将对象创建的管理与对象实例化混在一起。这样做错在何处?

2.工厂可以提高内聚性。原因是什么?

3.工厂还有助于测试。这就哪些方面而言是正确的?

观点与应用题

工厂的用途不仅仅是决定创建或者使用哪些对象，它们还可以解决模式带来的一个问题，从而有助于封装设计。请评估这一说法。

第21章 Singleton模式和Double-Checked Locking模式

21.1 概览

本章内容

本章继续讨论从第9章开始的电子商务案例研究。

在本章中，我们将：

介绍 Singleton（单件）模式，阐述工厂如何利用该模式控制一个存在的对象的实例数目；

给出Singleton模式的关键特征；

介绍Singleton模式的一个变体——Double-Checked Locking（双重检查锁）模式；

介绍我在实践中使用Singleton模式的一些经验。

Singleton模式和Double-Checked Locking模式都比较简单，而且非常常用。它们都可以用于确保某个类只有一个对象实例化。两个模式的区别在于：Singleton模式用在单线程应用程序中，而Double-Checked Locking模式用于多线程应用程序[3]。

在阅读本章时，请记住前一章中讨论过的工厂的各种概念。本章中，工厂的焦点放在封装用来控制对象数目的规则——在这里，只能有一个对象存在。模式有助于将使用和构造分离。

21.2 Singleton模式简介

意图，按照《设计模式》一书的说法

按照《设计模式》一书的说法，Singleton模式的意图是：保证一个类仅有一个实例，并提供一个访问它的全局访问点[4]。

Singleton模式的工作原理

Singleton模式的工作原理是：用一个特殊方法来实例化所需的对象。其中最关键的就是这个特殊方法。

调用这个方法时，检查对象是否已经实例化。如果已经实例化，该方法仅返回对该对象的一个引用。如果尚未实例化，该方法实例化该对象并返回对此新实例的一个引用。

为了确保这是实例化此类型对象的唯一方法，我将这个类的构造函数定义为保护或者私有的。

……所有协作对象都使用同一实例

Singleton 模式的本质在于，应用程序中的每个对象都使用 Singleton 类的同一实例，而不用必须负责将该实例四处传递给所有要使用它的对象。在一个协作对象对此实例所做的修改需要为另一个协作对象所见时，这一点尤其重要。

21.3 将Singleton模式应用到我们的案例研究

在第9章，将缴税规则封装在策略对象中。我们必须为每种可能的税额计算规则派生一个 CalcTax 类。这意味着需要反复使用同样的对象，只是轮流使用它们。

一个引出问题的例子：税额计算策略对象实例化一次且仅在需要时实例化

由于性能的原因，我可能不希望反复地实例化、然后再销毁这些对象。而且，虽然可以在开始时实例化所有可能的策略对象，但如果策略越来越多，这样做的效率将很低。（别忘了，整个应用程序中可能还有许多其他策略。）相反，最好按需要实例化，而且只进行一次。

问题在于，我不希望创建另一个对象来记住实例化了哪些对象。相反，我希望这些对象（即策略对象）自己负责只实例化一次。

Singleton使对象自己负责

这正是 Singleton 模式的意图。它使我们只对对象实例化一次，无需客户对象再关心对象是否存在。

Singleton 模式可以用例 21-1 所示的代码来实现。在这个例子中，我创建了一个方法（getInstance），它最多只实例化一个 USTax对象。通过将构造函数设置为私有的（即其他对象无法访问它），Singleton 模式防止其他任何人直接实例化USTax对象。

例**21-1 Java**代码片段：**Singleton**模式

```
public class USTax extends Tax {
   private static USTax instance;
   private USTax() { }
   public static USTax getInstance() {
      if (instance== null) instance= new USTax();
      return instance;
   }
}
```

具有多态的Singleton

然而，在这个案例研究中，实际上需要由SalesOrder对象询问Tax对象应该使用哪个Tax对象。这是因为SalesOrder不想了解存在哪个特定的Tax对象。这里组合了两个概念。

1.隐藏使用哪个具体类（Tax类中方法内的工厂）。

2.隐藏每个类进行了多少次实例化（USTax和其他 Tax具体类中的Singleton）。

如例21-2所示。

例**21-2 Java**代码片段：电子商务系统背景下的**Singleton**模式

```
abstract public class Tax {
   static private Tax instance;
   protected Tax() { };
   abstract double calcTax(
      double qty, double price);
   public static Tax getInstance() {
      // 这里创建了USTax对象。
      // 但你要根据相应规则创建自己对象
      instance= USTax.getInstance();
      return instance;
   }
}
public class USTax extends Tax {
   private static USTax instance;
   private USTax() { }
   // 注意下面的代码，由于我使用了相同的getInstance名称，
   // 所以要把USTax改成Tax
   public static Tax getInstance() {
      if (instance== null) instance= new USTax();
      return instance;
   }
}
```

SingIeton模式：关键特征

意图

希望对象只有一个实例，但没有控制对象实例化的全局对象。还希望确保所有实体使用该对象相同的实例，而无需将引用传给它们。

问题

几个不同的客户对象需要引用同一对象，而且希望确保这种类型的对象数目不超过一个。

解决方案

保证一个实例。

参与者与协作者

Client对象只能通过getInstance方法创建Singleton实例。

效果

Client对象无需操心是否已存在Singleton实例。这是由Singleton自己控制的。

实现

添加一个类的私有的静态成员变量，引用所需的对象（初值为null）。

添加一个公共静态方法，它在成员变量的值为 null时实例化这个类（并设置成员变量的值），然后返回该成员变量的值。

将构造函数的状态设置为保护或私有，从而防止任何人直接实例化这个类，绕过静态构造函数机制。

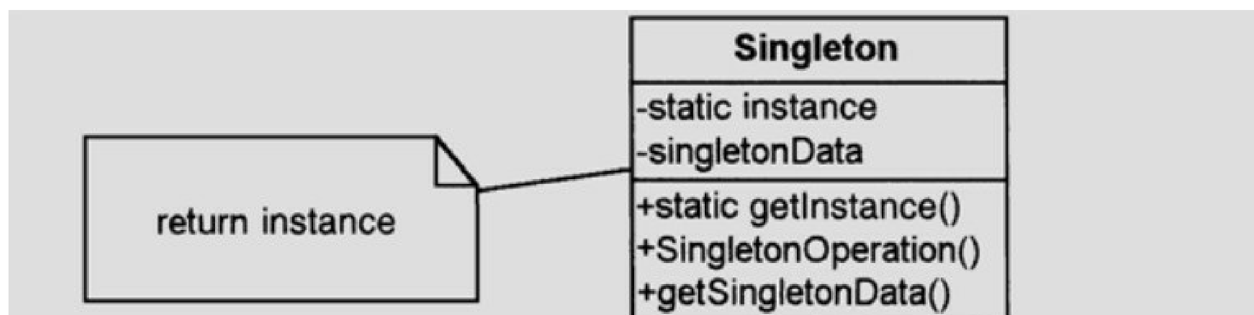


图21-1 Singleton模式的通用结构图

21.4 一种变体：Double-Checked Locking模式

只适用于多线程应用程序

Double-Checked　Locking模式仅适用于多线程应用程序。没有涉及多线程应用程序的读者，可以跳过这一节。本节假设你对多线程问题——包括同步有基本的了解。

多线程模式下，Singleton模式不能总是正常工作

在多线程程序中，Singleton模式可能会出现一个问题。

假设对　getInstance()方法的两个调用几乎同时发生，这种情况可能非常糟糕。考虑这时会发生什么。

1.第一个线程检查实例是否存在。因为实例不存在，该线程执行创建第一个实例的代码部分。

2.然而，假设在实例化完成之前，另一个线程也来检查实例成员变量是否为　null。因为第一个线程还什么都没有创建，实例成员变量仍然等于null，所以第二个线程也执行了创建一个对象的代码。

3.现在，两个线程都执行了Singleton对象的new操作，因此创建了两个对象。

这会出什么问题吗？可能会，也可能不会。

没问题，小问题，较差还是很严重

如果Singleton完全没有状态，这可能不是什么问题。

如果Singleton有状态，而且我们希望当一个对象改变其状态时，所有其他对象都应该看到，那么这就成了一个严重的问题。和第一个线程互操作的对象，与和其他所有线程互操作的对象都不同。可能会出现的问题包括：

使用不同Singleton对象的线程之间的状态不一致；

如果对象创建一个连接，实际上将是两个连接（每个对象一个）；

如果使用了计数器，将有两个计数器。

在 C++中，无论状态有无，程序都会产生内存泄漏，因为可能在实际上创建了两个对象的情况下，只销毁了其中一个对象。

发现这些问题可能非常困难。首先，双重创建是间歇性的——通常不会发生。其次，出现问题的原因可能很不明显，因为只有一个客户对象包含其中一个 Singleton 对象，而其他所有的客户对象引用的都是另一个Singleton对象。

同步化Singleton对象的创建

乍看来，只需同步化对 Singleton 对象是否已创建进行检查的操作即可。唯一的问题在于：这种同步化最后会导致严重的瓶颈，因为所有线程都必须等待关于对象是否存在的检查。

可能还可以将一些同步化代码放在 if (instance== null)判断之后。这也不会有什么用处。因为有可能两个调用都满足null检查的条件，然后再尝试同步化，最后还是会创建两个Singleton对象，一次创建一个。

一个简单的解决方案：Double-Checked Locking模式

解决的办法就是在检查到null之后进行“同步”（doSync()），然后再检查一次，确保实例尚未创建。如例21-3中所示。这称为Double-Checked Locking模式[5]。该模式的意图是优化掉不必要的锁定。这种同步检查最多进行一次，因此不会成为瓶颈。

Double-Checked Locking模式的特点如下：

在创建对象之前，添加一次检查，避免不必要的锁定。

支持多线程环境。

例**21-3 Java**代码片段：只实例化一次

```
public class USTax extends Tax {
  private static USTax instance;
  private USTax() { }
```

```
    public static Tax getInstance() {
        if (instance== null) {
            synchronized(this) {
                if (instance == null)
                    instance = new USTax();
            }
        }
        return instance;
    }
    private synchronized static void doSync() {
        if (instance == null)instance = new USTax();
    }
}
```

糟糕的是，这也不奏效

当Double-Checked Locking模式从C++转换成Java时，很快就会发现它对Java并不适用[6]。原因在于，Java编译器本身的优化工作会在构造方法实例化对象之前从构造方法返回指向该对象的引用。因此，在USTax对象真正完全构造之前，doSync就可能完成了。这会带来问题。而且，优化编译器会“注意到”“实例”成员“没有办法”在两个if语句之间改变状态，所以会优化掉第二个。在C#中使用volatile关键字就可以避免这一问题。

一个Java解决方案

解决这一问题并不是非常困难——我自己就能提出了一种方案：利用类装载程序，如例21-4所示。

例**21-4 Java**代码片段：只实例化一次

```
public class USTax extends Tax {
    private static class Instance {
        static final Tax instance= new USTax();
    }
    private static USTax instance;
    private USTax() { }
    public static Tax getInstance() {
        return Instance.instance;
    }
}
```

这个方案之所以奏效，是因为内部类（Instance）将只被装载一次，所以只会创建一个USTax对象。

21.5 反思

在一种语言中有效的技术未必在另一语言中适用

在最初所描述的Double-Checked Locking模式被发现无法在Java中工作时，许多人将其视为模式言过其实的一个证据。我认为恰恰相反。这里的错误在于看待模式的视角不对。批评者从实现的视角看待模式，所以在具体技术不能普遍适用时会感到吃惊。他们似乎惊讶于一种语言中的某个惯用法无法直接应用到另一种语言。可是，模式在功能层次和规约层次的描述仍然是成立的，只是实现层次有些区别而已。

事实上，即使在实现层次，模式的存在也是利大于弊的。因为模式中定义了这种惯用法后，开发人员有了共同的参照系，和可以用来讨论问题的共同语言。他们会迅速而且简便地了解到这个模式在 Java 中的工作方式与C++中不同，然后他们可以想出符合模式需求而且行之有效的解决方案。而且，其他语言中也已经设计出了替代方案，一个有力的

证据就是C#中volatile方案很快得到了采纳。

21.6 实践注记：使用Singleton模式和Double-Checked Locking模式

只在需要时使用

如果知道需要一个对象，而且没有性能因素需要将对象的实例化推迟到需要时进行，用静态成员变量包含对该对象的引用，往往更简单。

通常无状态

在多线程应用程序中，Singleton对象通常必须是线程安全的（因为这个对象将为多个对象所共享）。这意味着该对象要么在不同线程之间使用同步，要么没有数据成员（即只有作用域不能超过方法的变量）。

有状态的 Singleton本质上都是全局的，用来实例化其他模式中所用的对象

有状态 Singleton 应该谨慎使用，因为这样的全局数据可能会使系统的不同部分耦合。

无状态Facade可以用一个Singleton实例化，以避免出现一个以上Facade。Strategy模式的具体策略对象（无状态时），Bridge模式的具体实现（也是无状态时）也可以这样实例化。这对许多其他模式中的对象也是适用的。

可以修改为包含一个以上的对象

如果需要特定数量的实例，静态公共方法可以修改为包含一个以上的元素，当然，这样它们就不再是Singleton了。我们将在第22章中讨论这一点。

21.7 小结

本章内容

当需要确保某个类只有一个实例时，Singleton模式和Double-Checked Locking 模式是很常用的。Singleton 模式用于单线程应用程序，而Double-Checked Locking模式用于多线程应用程序。

这些模式说明了工厂可以是包含对象创建规则的单独对象或者方法。它们还说明，工厂并不仅仅与对象是否存在有关，而且涉及存在多少个对象。

复习题

简答题

1.Singleton属于哪一种模式类型？

2.Singleton模式的意图是什么？

3.Singleton负责创建多少个对象？

4.Singleton 使用一个特殊方法来实例化对象。这个方法有什么特殊性？

5.使用Singleton模式和Double-Checked Locking模式时的区别何在？

阐述题

让对象负责处理自己的单一实例化比全局地处理更好吗？为什么？

观点与应用题

1.你认为为什么《设计模式》一书称这个模式为“Singleton”？这对于它的作用而言是一个合适名称吗？为什么？

2.在最初所描述的 Double-Checked Locking 模式被发现无法在 Java 中工作时，许多人将其视为模式言过其实的一个证据。我却认为恰恰相反。你同意这里的逻辑推断吗？为什么？

第22章 Object Pool模式

22.1 概览

本章内容

本章讨论Object Pool（对象池）模式。这次我们不再使用国际电子商务案例，而是通过一个几年前开发的实际项目为例阐述。这个项目证明了我在第10章中提出的一个观点，即如果你理解了模式的原则，遇到可以应用未知模式的项目时，自己都有可能将模式推演出来。具体到这个项目，我最终从设计模式的基本原则中推出了Object Pool模式。后来我发现，我写出的模式别人也已经发现了，并且命名为Object Pool模式。我想向读者强调的一点是，理解设计模式的基本原则比记住一大堆图和模式，或者拥有一本大部头的参考书更重要。知道如何用模式思考，将使你更可能找到解决方案，或者推出适合于具体情况的模式。

在本章中，我们将：

描述Object Pool模式；

说明如何使用模式关注项目最重要的方面——将不会带来风险的事情延后；

说明工厂如何不仅能够管理对象，而且是描述对象正确操作的逻辑的最佳场所；

说明模式和理解如何封装问题为什么能够促进敏捷编程实践。

22.2 一个需要对对象进行管理的问题

连接池

几年前，我与一个公司签约，参与开发一个基于 Web 的个人投资系统。虽然这种系统今天已经稀松平常，那时还是新鲜事物。这种应用程序的大致目标是使用户能够通过万维网查看自己的个人投资情况，输入订单，对其账号中的股票、债券等等进行买卖。有关用户账号的信息保存在一台大型计算机中。物理架构如图22-1所示。



图22-1 个人投资系统的多层架构

所用的前端是用Java servlet编写的；它为我用C++编写的中间层提供数据。中间层与大型机通信以获取用户的投资信息或者提交订单。与大型机通信的唯一方式是使用大型机支持公司制定的一个特殊消息协议通过TCP/IP 连接。我编写的中间件代码负责获取用户的输入，并验证其有效性。它要查询存放所有信息的大型机才能完成这一操作。servlet前端本质上只是完成格式化和基本的检查。

典型的用户对话如下：

层：浏览器		层：中间件		层：大型机
用户登录	→	中间件获取用户 ID 和密码 格式化该信息并通过 TCP/IP 发送给主机，查看是否有效	→	大型机接受请求，检查有效性。相应地做出响应
用户请求查看自己的投资组合	→	中间件验证这是否一个有效的登录用户。如果是，将请求传给大型机	→ ←	大型机查询用户的投资组合，发送回中间件应用程序
显示信息	←	中间件获取大型机的响应，发送回前端		

吞吐量是主要关注点

这个系统的性能有一些特殊。我被告知需要更关注吞吐量，而无需操心响应时间。因为我的中间件应用要接受许多人的请求，这一需求意味着我需要处理尽可能多的事务，但是无需担心某个线程通过系统的时间。

在此之前我从来没有编写过这样的应用程序。对于如何平衡负载我并无良策。我知道与大型机的TCP/IP连接需要不止一个，但是应该是多少个呢？非常肯定的是，正确的数量应该在2～100之间——一个太少了，而超过100不会有什么好处，因为可用带宽无论如何不能处理超过20个连接。

那么应该用多少个 TCP/IP 连接呢？虽然我清楚这一问题最开始是无法得到答案的，但是必须在项目结束之前确定。这对于我来说真是进退维谷。不知道如何开始，也没有时间进行大量的测试或者仿真。而且那时候我其实对TCP/IP了解得也不够多。这意味着TCP/IP连接是这一项目的高风险因素，可能还不仅仅如此，因为项目团队中没有人知道其中的风险到底多高。

我喜欢迅速解决高风险的问题。如果不能解决，我会尝试隔离它们，或者转化为风险较低的任务。为了解决 TCP/IP 连接问题，我知道需要尽可能找到一个端到端的解决方案，它可以接受来自浏览器的请求，执行一些逻辑，将请求传给大型机，获得响应，然后将信息发回给浏览器。在完成这些以后，风险就降低了，因为我应该已经显示了通过TCP/IP 连接通信的能力。

要用TCP/IP连接实现这个端到端方案，必须完成以下工作。

1.建立一个稳定的TCP/IP连接。

2.确定所用TCP/IP连接的数量。

3.建立平衡连接负载的方法。

4.处理TCP/IP连接上的错误。（针对一些已知的故障情况。）

所有这四点都是非常重要的。要考虑的风险问题如下所述。

1.我需要多长时间熟练掌握TCP/IP？

2.如果改变连接数量，对使用代码有什么影响？

3.负载平衡对使用代码有什么影响？

4.错误处理对使用代码有什么影响？

对于问题1，没有别的办法，必须迅速提高自己的TCP/IP技术水平。但是问题2到问题4可以通过将这些问题封装起来，与使用代码隔离来缓解。也就是说，主要业务逻辑实际上只需要处理一个 TCP/IP 连接。连接如何工作，有多少个连接，以及如何处理错误，都是那些“连接到连接”代码要处理的问题。如果能够将这些问题与业务逻辑隔离——如果业务逻辑能够忽略这些问题，那么就可以以后再修改相关代码，在解决了问题1之后，而且不会影响其他代码。这说明必须将它们都隐藏（即封装）起来。

对我来说这是一种通用方法：努力寻找办法隔离修改系统所产生的影响。我知道在找到正确解决方案时必须修改一些东西，因为很少能够一开始就找到正确方案。这种情况下，必须尽快，因为这对于减少调试和返工更加重要。

脑子里有了这种方法，加上对模式的一般理解，就可以开始了。

不要依赖于渺茫的希望

我知道，因为没有太大的希望能够猜测出正确的解决方案，必须使系统能够在不影响代码的情况下（除了一些可能的非常微小的改动之外）修改TCP/IP连接的数量。这意味着我的客户代码（使用TCP/IP连接的代码）不应该在连接数量改变时必须改变。对我来说，这意味着客户对象甚至不能知道有多少个TCP/IP对象。谁应该知道呢？其他对象！

创建一个TCP/IP管理器

这个所谓的“其他对象”可能是我的 TCP/IP 对象的管理器。因此这里有两个主要功能：

1.要与大型机通信的TCP/IP对象；

2.对这些对象的管理。

虽然可以将所有这些逻辑都放在 TCP/IP 对象本身里，但内聚原则指示我们应该用两个不同的对象。我称控制一个 TCP/IP 对象的对象为“端口（Port）”，因为 TCP/IP只是程序如何与大型机通信的一种实现。虽然我现在使用的是TCP/IP（可能总是如此），但是在我的脑中，对象是通往大型机的端口。我实际上认为这种连接是用户信息的端口。很自然，我决定将我的端口管理器命名为PortManager。

尽管可能有很多端口，但只有一个PortManager。这个对象可以处理所有 Port。如果需要确保最多只有一个对象，应该用什么模式呢？当然是Singleton模式。我当时正是这样决定的。

Singleton确保了只有一个PortManager，我还希望这个PortManager是唯一能够创建Port的对象[7]。封装Port的管理之后，我可以任意选择使用多少个Port。我随便选择了5个。一开始选择任何数字都一样。

我实现了PortManager，如下所示[8]。

数据成员

给它定义了一个私有的包含5个Port引用的数组，使用一个常数定义这个数值。在 PortManager的构造方法中实例化这个数组。如果任何引用无法实例化，就抛出异常，因为这是个糟糕的开始：不能获得5个连接将无法很好地承担保持通信通畅的任务！

方法

有一个方法，可以调用来得到一个Port，称为getInstanceOfPort()。

另一个方法可以调用（用 Port的引用）来告知该 Port不再活动：returnInstanceOfPort(Port portToRelease)。

实现这两个方法是非常简单的。

getInstanceOfPort()第一个方法只需要循环遍历Port数组，找到第一个可访问端口。（每个端口都有一个状态表示是否在使用。）如果找到一个端口，其状态更新为活动，并返回它。如果没有找到，使线程休眠

一秒，然后再试。

returnInstanceOfPort(Port portToRelease)这个方法也不难。只需要遍历端口，直到找到给定的引用，并将其状态更新为不活动。如果找不到就抛出一个异常，因为不应该发生这种事情。

客户代码非常简单。当有消息需要处理时，客户将：

1.向PortManager请求一个端口；

2.按需要使用端口；

3.释放端口。

客户代码没有必要关心端口的创建，或者要用多少个端口。

模式有助于敏捷

注意到这里不仅使端口与 Port代码解耦了，而且还意味着我可以将许多工作推迟到项目后期，这一点非常重要。更好的是，在客户代码中，我还可以忽略错误处理，这是C++中与TCP/IP有关的较难问题之一。为什么能够忽略呢？因为我相信Port和PortManager要么能够完全处理错误，要么处理到这样的程度：客户代码在真正实现时不会受到实质性的影响。我大可以只关注问题的更重要的方面：客户代码。

关注最重要的问题对于敏捷开发至为关键。在极限编程中甚至为此创造了一句格言：YAGNI (ya ain’t gonna need it，即“你不需要它”)。它反映了这样的理念：应该构建现在需要的东西，而忽略其他。应该尽早完成最重要的工作，何时能够解决这些最重要的问题，会产生最大的影响。这还说明，应该避免开发那些分散精力而且一般用不上的（因此构建它们是浪费资源的）东西。

本例中，需要完成的最重要的工作就是中间件应用程序和大型机之间的通信。负载平衡和错误处理等工作，虽然也非常重要，但是会推迟我构建系统的进程。首先关注通信，将它与系统的其他部分解耦，就能够让其他人响应我的需求同时开展工作。其他需求可以以后再来处理。

我得到的就是Object Pool模式

当时我还不知道，自己最后推出了Object Pool模式。我能达到这样高质量的设计，完全是因为遵循了设计模式的原则。这也正是 Object Pool模式所支持的。虽然有读者可能会说，既然能够自己推出来，岂不是说明懂得设计模式并不是那么必要？我感觉了解模式能够确保在适用的时候使用。况且经验不足的设计师可能不会采取我的方法。从别人那里学习比完全自己开拓要省力得多。再说，在知道了这个模式之后，我就能够容易地与知道它的其他人交流我所选择的方法了，而且我完全相信我们在讨论同一件事情。

处理错误

使通信功能运行起来之后，我决定把注意力转移到错误处理上。这对我的代码的不同部分含义也是不同的。

1.对于Client，它意味着能够获得另一个Port以供使用和重发消息。

2.对于遇到错误的Port，它意味着禁用自己，而且不再尝试使用。

3.对于 PortManager，它意味着记住这个 Port已经出了故障，需要获取一个新的代替它。

已经实现的解耦使这一点变得非常容易。具体实现如下所述。

1.在Port出现错误时抛出异常。

2.在Client中消息开始传输的点捕获异常。查看有什么东西需要收回（通常没有）相对比较容易。然后，返回该 Port 给PortManager，说明它已经出了故障，请求另一个。

3.在PortManager中增加一个名为returnBadPort (Port badPort)的方法，这个方法负责将该 Port 标记为有问题的（与returnInstance- OfPort方法的工作方式相同）。然后从数组中删除该 Port，在其位置插入新的一个 Port。如果无法获得新的Port（这种可能性很小），将通过电子邮件向系统管理员发送一条消息——如果系统似乎有崩溃和损坏的迹象，这是非常可取的。

然后我会用剩下来的Port（如果还有的话）服务其他请求。

继续深入

这应该已经足够了，但是并非如此。当系统继续开发，看上去就要实际上线运行的时候，我又开始睡不着觉了。有些问题困扰我，而且挥之不去。此时，我用到的另一个设计原则是留意自己的直觉：你经常比自认为的知道得多！

困扰我的是即将通过我的管道进出的数以百万计的钱居然无人监管。虽然此前我编写了大量软件系统，其中有些承担的任务也非常关键，但是这个系统的压力似乎更大。如果这个系统哪天半夜出了问题，第二天凌晨发现后再重启系统将是无法接受的。数以百万计的交易金额可能没有执行。这种事情就是想一想都让人高兴不起来！必须有某种监管机制。

我知道必须解决这一问题。事实上，我觉得自己必须有双重保证。我必须确保系统健壮，然后还需要确认它总是会这样健壮。我记得曾经在Steve Maguire的著作Writing Solid Code[9]中读到对于代码中任务关键的部分应该有双重检查方法。这使我想到，如果有另外一个独立的机制来检查通信，就用不着担心了。

双重检查

答案相对要简单一些。PortManager已经负责创建Port了。我只需要再给它增加一个职责验证这些 Port的完整性。为此，我选择让它启动一个线程，每15分钟检查一次，检查将执行以下任务。

查看Port还有多少个未处理请求（如果数目非常大还需要采取其他措施）。

查询 Port 的状态（活动与否），与未处理请求的数目进行比较，查看是否有故障出现。

查看有多少Port处于错误状态（应该至少是1或者2）。

基本上，我可以把需要的任何东西放到这段代码中，如果需要还可以扩展。它能够提供更高的视角，随着我对可能故障的更多了解，这个

视角可以会不断改进。这样做不会显著影响代码的其他部分。

幸亏有双重检查，我又可以高枕无忧了！

22.3 Object Pool模式

虽然我使用Object Pool模式是为了说明如何使用工厂，以及模式如何有助于敏捷性，但是Object Pool模式本身在存在共享资源而且与该资源单点联系较为有益时，也是非常有用的。这种单一联系点（池）还可以履行其他职责（比如错误处理）。通过封装这些职责，使用这些对象的客户代码不仅能够免于这些职责，而且还可以与这些职责相关的修改隔离开来。

22.4 观察：工厂的作用不仅是实例化

实例化，管理，错误处理

在对工厂的讨论之初，我们说工厂是为了将使用与构造分离。现在已经说明了怎样分离使用和构造以及对象管理。我们甚至还扩展到包括了错误处理，因为存在有问题的Port将意味着需要新的Port。

关键在于，在我们开始将对象视为带有职责的事物时，如何分离这些职责就更加清晰了。在找到了职责之后，经常能很容易看出系统所应具有的正确结构。这将使代码更加清晰、健壮、更易管理和修改，而且我们也可以高枕无忧了。

Object Pool模式：关键特征

意图

在创建对象比较昂贵，或者对于特定类型能够创建的对象数目有限制时，管理对象的重用。

问题

对象的创建和/或管理必须遵循一组定义明确的规则集。通常这些

规则都与如何创建对象、能够创建多少个对象和在已有对象完成当前任务时如何重用它们等等相关。

解决方案

在需要一个 Reusable对象时，Client调用 ReusablePool的acquireReusable方法。如果池是空的，那么acquireReusable方法创建一个 Reusable对象（如果能够），否则，就等待直到有Reusable对象返回集合。

参与者与协作者

ReusablePool管理着Client所用的Reusable对象的可用性。Client然后在一个有限的时间段内使用Reusable对象的实例，ReusablePool包含所有Reusable对象，这样就可以对其以统一的方式进行管理。

效果

最适用于对对象的需求一直非常稳定的时候，需求变化太大会带来性能问题。Object Pool中为了解决这一问题，限制了能够创建的对象数量。使管理实例创建的逻辑与实例被管理的类分离，可以得到内聚更好的设计。

实现

如果可以创建的对象数量有限制，或者池的大小有限制，可以使用一个简单的数组来现池。否则，使用vector对象，负责管理对象池的对象必须是唯一能够创建这些对象的对象。ReusablePool 是用 Singleton 模式实现的。另一种变体是在Reusable对象中加一个释放方法——让它自己返回到池。

参考文献

这个模式并非出自《设计模式》一书。它是在Mark Grand的书Patterns in Java卷1中135～142页所描述的[10]。Clifton Nock的Data Access Patterns一书在数据库资源背景下非常详细地讨论了这一模式，其中称之为Resource Pool模式[11]。

图22-2 Object Pool模式的通用结构图

22.5 小结

创建，管理和错误处理

本章中说明了如何用工厂解决创建和管理对象重用之外的问题。Object Pool模式之所以非常有用，是因为两个原因：

说明了如何使用工厂实例化和管理对象；

说明了职责的封装如何有助于开发人员将注意力集中在最需要的地方。

复习题

简答题

1.设计时应该遵循的三个通用策略是什么？

2.Object Pool模式融合了哪两个模式？

3.Object Pool模式的意图是什么？

阐述题

在XP社区中YAGNI是什么意思？

观点与应用题

广泛阅读是一种重要的学习。有些知识说不定什么时候就会用到，比如Steve Maguire的著作Writing Solid Code就是一个好例子。从你自己的经历中举出一个例子，证明这一点。

第23章 Factory Method模式

23.1 概览

本章内容

本章将继续从第9章开始的电子商务系统案例研究。

在本章中，我们将：

通过讨论这个案例中增加的需求介绍Factory Method（工厂方法）模式；

描述Factory Method模式的意图；

描述Factory Method模式的关键特征；

描述Factory Method模式如何融入最新的面向对象语言；

介绍我在实践中使用Factory Method模式的一些经验。

23.2 案例研究的更多需求

新需求：实例化数据库对象的责任

在第19章中，我忽略了一个问题：如何对当前背景所需的数据库对象实例化。我可能不希望Client负责实例化数据库对象，相反，想将这个责任指派给QueryTemplate类自己。

第19章中，QueryTemplate类的每个派生类都专门针对某种数据库。因此，我想让每个派生类负责实例化与自己对应的数据库。无论QueryTemplate（及其派生类）是否是唯一使用数据库的类，都是如此。图23-1显示了这一解决方案。

Template Method使用 Factory Method模式

在图23-1中，Template Method模式中的doQuery方法使用makeDB方法实例化相应的数据库对象。QueryTemplate并不知道要实例化哪个数据库对象；它只知道肯定有一个数据库对象必须要实例化，而且要为其实例化提供一个接口。QueryTemplate的派生类需要知道要实例化哪个数据库对象。因此，在这个层次上，可以将如何实例化数据库的决策推迟到派生类的某个方法中。

因为有一个创建一个对象的方法，所以这种方式被称为 Factory Method（工厂方法）。

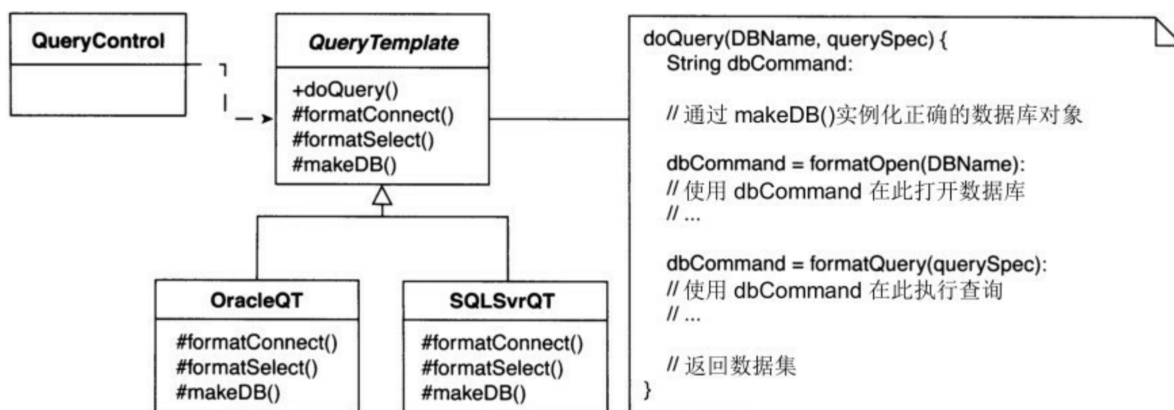


图23-1 使用Factory Method模式（makeDB）的Template Method（doQury）

公开方法还是保护方法？

请注意，makeDB方法是受保护的（图中用#符号表示）。在本例中，只有 QueryTemplate 类及其派生类可以访问这些方法。如果希望QueryTemplate 之外的对象能够访问这些方法，则这些方法就应该是公开的。这是另一种、很普遍的Factory Method模式的用法。在本例中，我仍然让派生类决定实例化哪个对象。

23.3 Factory Method模式

标准化步骤

Factory Method模式是一个旨在帮助创建责任分配的模式。按照《设计模式》一书的说法，Factory Method模式的意图是：

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使一个类的实例化延迟到其子类[12]。

23.4 Factory Method模式与面向对象语言

已经融入Java、C#和C++库

Factory Method模式已经在所有主要的面向对象语言中实现了。

在 Java 中，集合的 iterator方法是工厂方法。这一方法返回被请求集合的迭代器的正确类型。

在 C#中，集合实现了 IEnumerable 接口。这一接口定义了GetEnumerator方法，这是一个获取集合迭代器的工厂方法。

在C++中，用到的工厂方法包括begin()和end()。

所有这些情况下，用来获取正确迭代器的方法都使用了 Factory Method模式。

23.5 实践注记：使用Factory Method模式

Abstract Factory模式可以用一系列Factory Method模式来实现可用来绑定对应的类层次

在Abstract Factory模式的经典实现中，有一个抽象类来定义创建一系列对象的方法。为每个可能存在的对象系列都派生一个类。所有定义在抽象类中，然后在派生类中覆盖的方法都遵循了Factory Method模式。

有时候需要创建与已有的类结构对应的一个类层次结构，新的层次包含某些委托的职责。这时，保证原层次中的每个对象都能实例化这个

平行层次中对应的对象非常重要。为此目的可以使用Factory Method模式。在前面提到的语言例子中，Factory Method模式将不同的集合和与集合相关联的不同迭代器绑定起来，如图23-2所示。

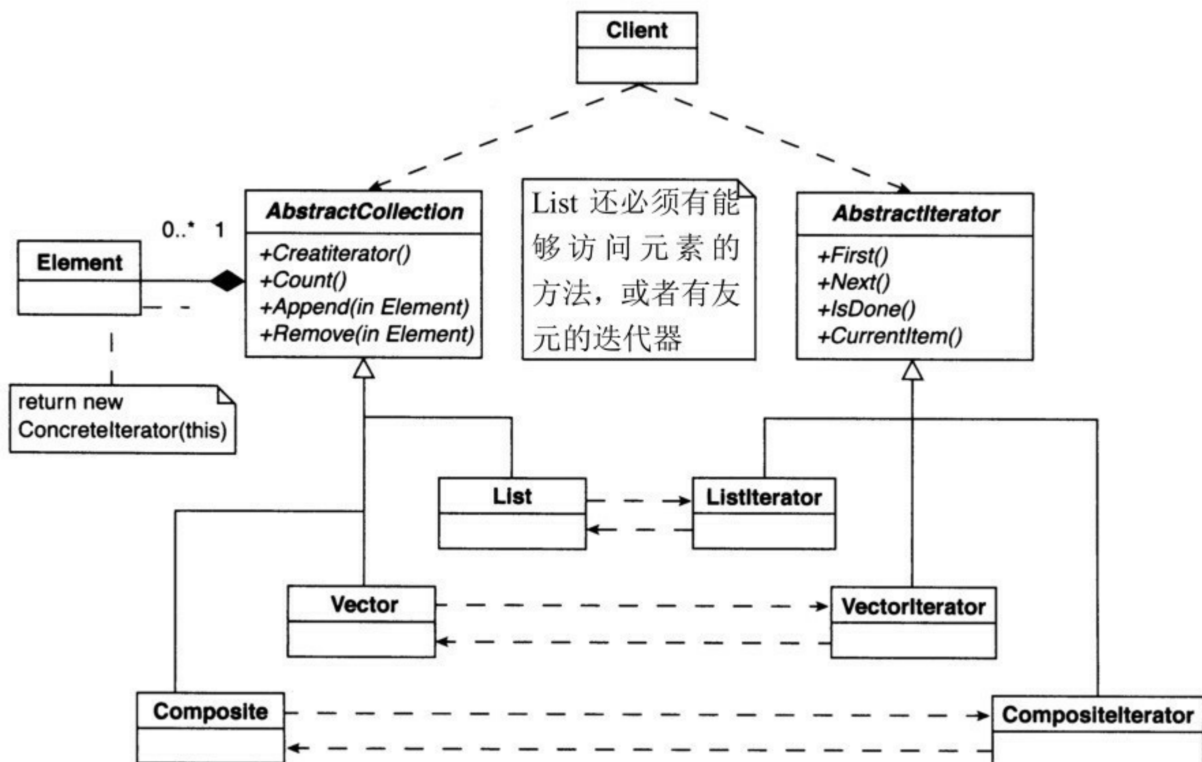


图23-2 绑定对应的类层次

Factory Method 常用于模式架中

Factory Method模式在定义框架（framework）时非常常用。这是因为框架处于抽象层次。通常它们不知道而且也不应操心具体对象的实例化。它们需要将有关具体对象的决策推迟给框架的用户。

Factory Method模式：关键特征

意图

定义一个用于创建对象的接口，让子类决定实例化哪一个类。将实例化推迟到子类。

问题

一个类需要实例化另一个类的派生类，但不知道是哪一个。Factory Method允许派生类进行决策。

解决方案

派生类对实例化哪个类和如何实例化做出决策。

参与者与协作者

Product是工厂方法所创建的对象类型的接口。Creator是定义工厂方法的接口。

效果

客户将需要派生 Creator，以创建一个特定的 ConcreteProduct对象。

实现

在抽象类中使用一个抽象方法（即 C++的纯虚函数）。需要实例化一个被包含对象的时候抽象类的代码将引用此方法，但是不知道需要的对象是哪一个。

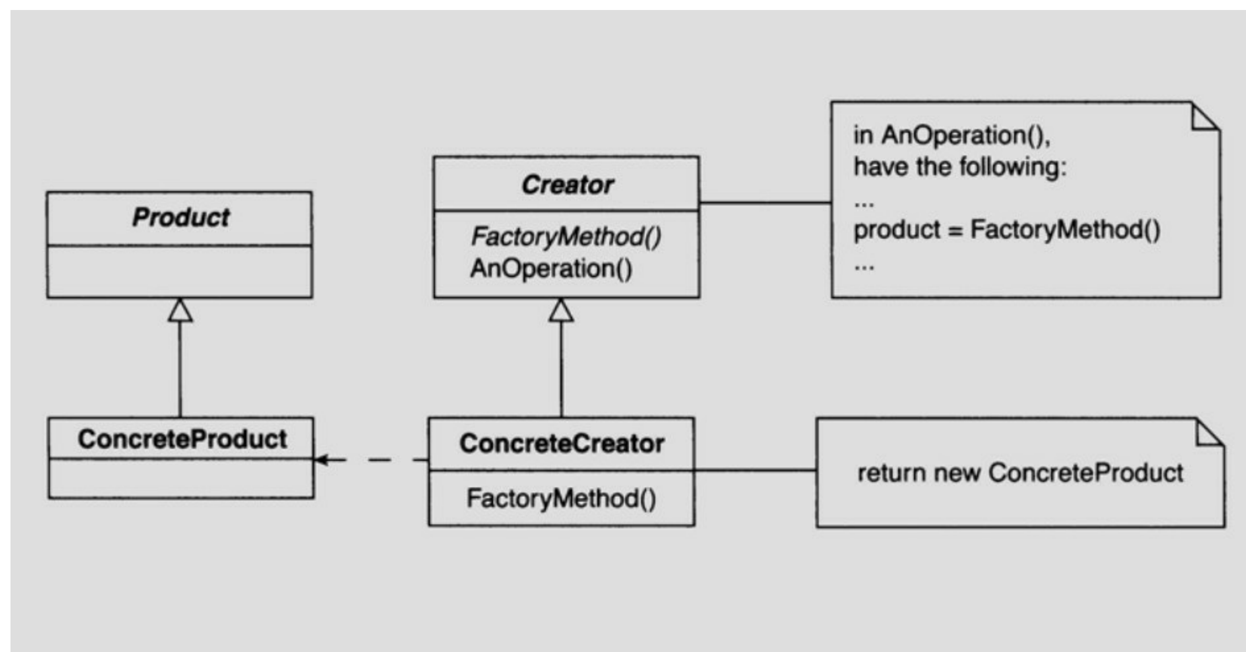


图23-3 Factory Method模式的通用结构图

23.6 小结

本章内容

Factory　Method模式是一个很简单的模式，我们会不断用到。它可以用于需要将对象实例化的规则推迟到某个派生类的情况。在这种情况下，将方法的实现放在负责该行为的对象中，最自然。

复习题

简答题

1.工厂应该负责什么工作？

2.使用Factory Method的主要原因是什么？

3.Factory　　Method已经在所有主要面向对象语言中得以实现。在Java、C#和C++中它是如何`提供的？

阐述题

1.为什么这个模式被称为“Factory Method”？

2.Factory Method模式与其他工厂模式是如何配合的？

3.《设计模式》一书中说，Factory　Method的意图是“定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory　Method使一个类的实例化延迟到其子类。”为什么这很重要？

观点与应用题

1.你怎样着手决定一个方法应该是公开、私有还是保护的？

2.本章的篇幅比较小，但是讨论的模式却很重要。请举出可以使用这个模式的例子。

第24章 工厂模式的总结

24.1 概览

本章内容

本章将总结本部分中我们所得到的教益。还会讨论创建和使用对象的三个主要任务。

1.根据所履行的责任找出对象。这时共性和可变性分析极为有用。

2.决定如何使用这些对象。主要是考虑对象间的关系，这正是许多模式所要解决的问题。

3.决定如何管理这些对象。这正是工厂的用武之地。

还将总结将软件开发任务分为几个不同步骤，每一步骤只解决以上一个问题的优点。

24.2 软件开发过程中的步骤

分而治之

长期以来，人们已经公认，开发软件时将代码分为多个模块对于提高所生成的代码的质量有极大的好处。模块更容易管理，而且在设计得当时也更容易修改或扩展。但是，在功能分解的系统中，模块主要用于模块化不同的功能。而在面向对象系统中，出现了一种新的可能。第1章讨论了三种不同的视角：概念视角、规约视角和实现视角。本章讨论设计应用程序时可以使用的另外一组视角：使用视角和创建/管理视角。

这样做的威力来自这样的事实：概念上相似的对象从使用的视角来

看，可以以同样的方式处理。然而，创建对象时，负责创建的实体通常需要了解要创建的是哪个具体对象，以及何时创建这个而非那个对象的规则。在设计系统时，让最复杂的部分在概念层次使用其他对象是最有用的。这意味着要遵循开闭原则、依赖倒置原则和 Liskov 替换原则。但是，要实现所有这些，使用对象就不应该知道所用的是哪一个特选对象，因此，需要有对象来负责此事——工厂就出现了。

将工厂称为“对象管理者”可能更合适，因为创建对象只是它们的责任之一。通过封装对象的创建和其后的管理，更复杂的使用对象就无需处理这些问题了。这意味着出现新功能的需要时，可以在已有系统的背景下处理，开发出新功能。如果概念上还没有它的位置，那就在系统中重构出一个，然后再添加新代码。

这是一个两步方法。首先，为新代码重构一个位置，然后将其加入系统。这种方法的好处有以下几个方面。

这意味着我们总是有能够工作的系统。

小步前进意味着调试工作减少了。

集成新代码的成本（大多数人都认为这是扩展代码成本最高的地方）始终很低。

代码的质量不会降低，而且设计不会变坏。

24.3 工厂与极限编程实践殊途同归

重构的正确方式

重构可以用来修改劣质代码或者作为扩展优秀代码的方式。使用重构来添加新功能的正确过程如下所述。

重构已有代码，以配合新代码。（没有增加任何新功能。）也就是说，对于关注区域中不符合开闭原则的代码，要重构使其符合。

然后加入新功能。

在工厂的背景下，这意味着要这样编写系统：使用对象不知道它在使用哪个特定的实现。如果什么地方不符合，而且完成该功能的方法不止一种，那就重构代码达到这一点。完成以后，添加新功能就只需要编写出来（这往往是避免不了的），修改负责这些类型对象的工厂/管理对象。

24.4 系统的扩展性

本质优点

这种方法的优点在于，它对于各种尺度的系统都行之有效。在开始，我们的选择很少（如果说还有的话），工厂可能是类自己的封装了构造操作的方法。接下来，随着系统越来越复杂，我们可能编写专门的工厂/管理对象，在其中写入正在讨论的规则。最终，可能需要使用数据库或者配置表体现规则。

无论怎样完成，工厂/管理逻辑都是封装在方法和对象之后的，从而将这些规则与使用方软件分离。如果逻辑变得非常复杂（这是常事），它与系统仍然是松散耦合的，可以保持灵活性和易扩展性。

软件中总是会发生变化。无论何种变化都或者会影响对象、服务的用户，或者影响实例化对象的工厂。保持分离，就减少了维护工作量，只需维护这些实体中的某一个，很少会是两者。

这又归结到一个基本原则，对于系统中的任意两个实体A和B，应该将它们之间关系限制为A使用B，或者A创建/管理B，但是两种关系永远不要同时存在！

---

[1].即工厂模式。——译者注

[2].本书中，“使用对象”与“客户对象”是同义词，“工厂对象”和“创建对象”可以互换。——译者注

[3].如果你不知道什么是多线程应用程序，不用担心。要理解基本原理，你只需关注Singleton模式就行。

[4].Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.127。

[5].Martin R.、Riehle D.和Buschmann F.，Pattern Language of Program Design，Boston:Addison-Wesley，1998，p.363。

[6].请参见本书网站http://www.netobjectives.com/dpexplained了解更多信息。

[7].因为那时我使用的是C++，所以将Port构造函数设为保护的，而PortManager设为它的友元。如果使用C#，就可以使用委托处理这种限制了。参见本书的配套网站http://www.netobjectives.com/dpexplained了解如何实现。虽然不那么显而易见，但是很容易。如果使用Java，可以将PortManager和Port类放在一个包中。

[8].请注意虽然开发这个项目时我当时使用的是 C++，但为了和本书其他部分一致，此处所用的代码是Java的。

[9].Maguire S.Writing Solid Code，Redmond，WA:Microsoft Press，1993，p.33。

[10].Grand Mark.，Patterns in JavaVolume 1，A Catalog of Reusable Design Patterns Illustrated with UML，New York:John Wiley&Sons，Inc.，1998，p.135-142。

[11].Nock Clifton.，Data Access Patterns，Boston:Addison-Wesley，2003。

[12].Gamma E.、Helm,R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston:Addison-Wesley，1995，p.107。

第八部分 终点与起点

概览

本部分内容

本部分将继续讨论面向对象设计的新视角。具体而言，本部分讲述设计模式在设计和实现中如何使用这种视角。最后是对进一步阅读的推荐书目。

章 讨论的主题

25 设计模式回顾：总结与新起点

在这种面向对象设计新视角的大背景下考察设计模式的动机和关系

26 参考书目

推荐图书和其他资源供未来学习之用

第25章 设计模式回顾：总结与新起点

25.1 概览

本章内容

在任何书的最后，回顾一下所得总是很好的。本书中，我们尝试通过一种可能非常新颖的方法，使你更好地理解面向对象原则。其中教授了设计模式，并使用设计模式说明设计模式是怎样解读面向对象范型的。设计模式回答了一个基本问题："为什么以这种方式设计？"。

在本章中，我们将：

看待面向对象原则的新视角，以对设计模式的理解为基础；

设计模式如何帮助我们封装实现；

共性和可变性分析，如何与设计模式一起帮助我们理解抽象类；

按照所涉及的责任对问题域进行分解；

设计模式与从背景设计；

模式之间的关系；

设计模式与敏捷编程实践。

最后，我提供了一些来自亲身经验的实践注记。

25.2 面向对象原则的总结

从新视角看对象

在讨论设计模式的过程中，我们已经说到了许多面向对象范型的原则。这些原则可以总结如下。

对象是具有明确定义的责任的事物。

对象对自己负责。

封装指的是任何形式的隐藏：

数据隐藏；

实现隐藏；

类隐藏（在抽象类或接口后）；

设计隐藏；

实例化隐藏。

使用共性和可变性分析抽象出行为和数据中的变化。

按接口设计。

将继承看成一种将变化概念化的方法，而不是创建已有对象的特殊情形。

将变化放入一个类中，并与该类中的其他变化解耦。

力求松耦合。

力求强内聚。

将使用一个对象的代码与创建该对象的代码分离。

在应用“一次且仅一次”规则时要绝对小心。

通过“按意图编程”，使用反映意图的名字，确保代码的可读性。

在编程之前就考虑代码的可测试性。

25.3 设计模式如何封装实现

隐藏细节中的变化

已经介绍的设计模式中有几个都具备这样的特点：对客户对象屏蔽了实现细节。例如，Bridge 模式对客户对象隐藏了如何实现 Abstraction 的派生类的细节。此外，Implemention接口也对Abstraction及其派生类隐藏了实现。在Strategy模式中，每个ConcreteStrategy类的实现都隐藏了。《设计模式》一书描述的大多数模式都是如此：它们提供了隐藏具

体实现的各种方式。

隐藏实现的价值在于，模式使开发人员能够容易地添加新的实现，因为客户对象不知道当前实现的具体工作细节。

25.4 共性和可变性分析与设计模式

共性和可变性分析

第10章说明了如何用共性和可变性分析推演出Bridge模式。许多其他的模式也可以这样推演出来，包括Strategy、Iterator、Proxy、State、Visitor、Template Method、Decorator、Composite以及Abstract Factoy。但是，更重要的一点是寻找共性（使用共性和可变性分析）本身有助于发现问题域中存在的模式。

例如，在Bridge模式中，可以从几种特殊情况开始。

用绘图程序1（DP1）绘制一个正方形。

用绘图程序2（DP2）绘制一个圆。

用绘图程序1（DP1）绘制一个矩形。

了解Bridge模式，有助于发现这些特殊情况中存在着两处共性，都有：

绘图程序；

要绘制的形状。

Strategy 模式与此类似，看到若干不同的规则时，我就知道应该寻找这些规则之间的共性，从而将它们封装起来。

虽然我们只用共性和可变性分析就可以推出许多模式，但是不断学习模式、阅读相关文献仍然非常重要。模式为讨论分析和设计中所得教益提供了背景；模式为开发团队提供了讨论问题的通用词汇表；模式使我们能够将最佳实践方法应用于代码之中。

25.5 按责任分解问题域

共性和可变性分析的下一步

共性和可变性分析可以辨识出概念视角（共性）和实现视角（每个具体的变化）。如果只考虑共性和使用共性的对象，可以以另一种方式思考问题——分解责任。

例如，在Bridge模式中，模式告诉我们将问题域看作由两个不同类型的实体（抽象和实现）组成。所以，没有必要限于只是进行面向对象的分解（也就是将问题域分解成多个对象），也可以尝试将问题域分解成责任，如果这样更加容易的话；然后可以定义必需的对象来实现这些责任（最终还是达到了对象分解的目的）。

将这种方法扩展一下，就是前面讲过的一条规则：设计师不应该在了解所需的所有对象之前操心对象的实例化。这一规则可以看成是将问题域分解成两部分：

需要哪些对象；

如何实例化和管理这些对象。

各种具体的模式往往有助于思考如何分解责任。例如，在需要将问题域分解成总是要用的主要责任（即 ConcreteComponent）和可能会有的变化（即 Decorator）时，Decorator 模式为我们提供了一种灵活地组合对象的方法。Strategy模式则将问题分解成使用规则（无论使用什么规则）的对象和规则本身。

25.6 模式和从背景设计

模式是从背景设计的微观示例

在本书前面的CAD/CAM问题中，我说明了如何通过关注设计模式之间形成的背景关系来使用设计模式。从背景设计是第14章中讨论的依

赖倒置原则的另一种叙述方式。设计模式共同协作有助于应用程序架构的开发，而且可以用于区分有多少模式是从背景设计的微观示例[1]。例如：

Bridge 模式告诉我们在 Abstraction 的派生类的背景下定义Implementation类；

Decorator模式让我们在原组件的背景下设计Decorator类；

Abstract Factory模式让我们在整个问题的背景下定义对象系列，从而可以看出需要实现哪个对象。

按接口设计就是在背景设计

事实上，从一般意义上说，按接口设计和多态就是一种从背景设计。如图25-1所示，这是图8-5的复制。请注意抽象类的接口定义了背景，所有派生类都必须在此背景中实现。

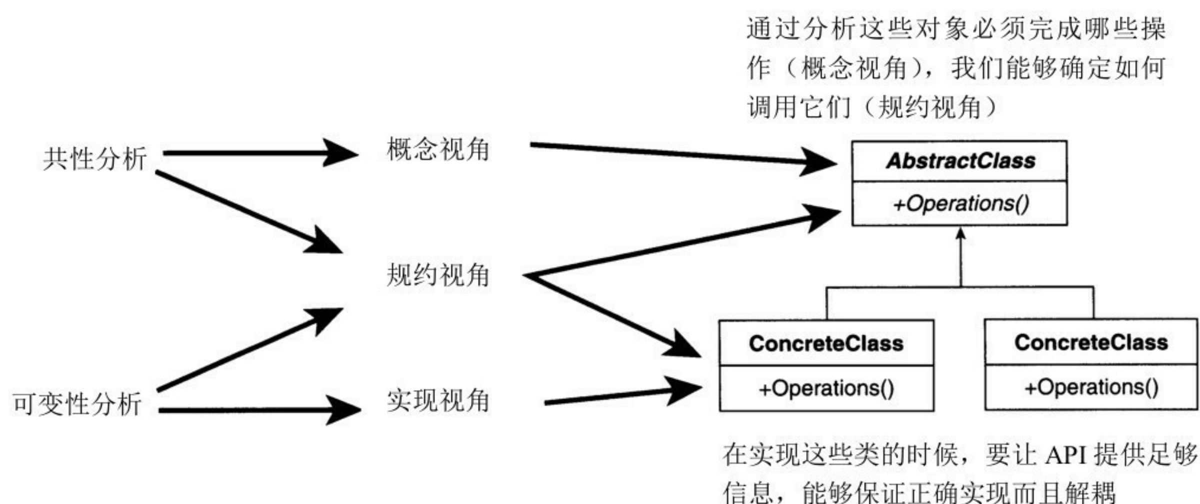


图25-1 共性和可变性分析、三种视角与抽象类之间的关联

25.7 模式内部的关联

模式并不是真正重要的

需要老实交待的是，我曾经在设计模式课堂上用Alexander的某些话来开玩笑。在用了大半天谈论模式有如何如何好之后，我拿起

Alexander的Timeless Way of Building一书，翻到最后，说：

这本书有 549 页。在第545 页，显然，非常接近结尾了，Alexander这样说："在这最后阶段，模式不再重要……"[2]

我停下来，说："我真希望他在一开始就告诉我们这一点，这样我们就省下许多时间了!"在学员们还没有起来造反之前，我继续引用书中的话："模式已经教会了你对真实的感悟力。"[3]

我最后是这样说的："如果你读过Alexander的书，你将知道什么是真实的——模式所描述的关系和约束因素。"

模式为我们提供了谈论这些内容的方法，但是，模式本身并不是最重要的。软件模式也是如此。

软件模式说明了多个方面的因素

每个模式都描述了某个特定背景中一个特定问题的约束因素、动机和关系，并为我们提供了一种解决这些问题的方式。例如，Bridge模式描述了一个抽象的派生类和可能的实现之间的关系。Strategy 模式则描述了以下几项之间的关系。

使用一组算法的一个类（Context）。

这组算法中的成员（Strategy）。

Client——使用Context对象并指定要用的算法。

模式的各个部分如图25-2所示。



“知识体系其实并非设计模式的组成部分，它基于设计模式、所用的语言、特定的环境等。”

图25-2 模式的各个部分

25.8 设计模式与敏捷编程实践

硬币之两面

有些人对模式的理解完全是望文生义，误解了模式的本质。他们认为模式就是放之四海皆准、千篇一律的解决方案。因此，模式似乎是与敏捷编程技术背道而驰的。但是，我希望本书已经说明模式绝非他们想象的那样。考察模式背后的理念和原则，可以看到它们对于敏捷编程实践（包括极限编程和测试驱动开发）是非常有用的。

我们已经说明了模式能够帮助引导我们进行重构。我们还讨论了模式与代码的可测试性的内在一致性。良好的编程和设计技术往往都不会相互抵触。即使乍看上去有些矛盾，稍加深入研究，就会发现它们都是殊途而同归。

25.9 实践注记

从一本书中完全学会模式是不可能的。必须编写代码，在设计中使

用它们。将模式编码实现是很容易的。假想出一个示例，实现模式的设计是很好的第一步。但是，请记住，这并非真正的模式，而只是它的一个方面而已。当然，它有助于你理解其他方面。

可采取的方法

在学习模式的过程中，寻找以下约束因素和概念会有所帮助。

这个模式隐藏了什么实现？这样我们就可以修改它。

这个模式中有什么共性？这有助于你找到共性。

这个模式中对象的责任是什么？这可以更容易地按责任进行分解。

这些对象之间有什么关系？这将提供这些对象的约束因素的信息。

这个模式本身怎样成为从背景设计的微观示例？这使我们能够更好地理解为什么这个模式是优秀设计。

开发软件时，看看是否这些问题的答案与正在解决的问题域有关。如果有关，看看模式中是否有什么可以派上用场。

25.10 小结

本章内容

本章中总结了面向对象设计的新视角，叙述了设计模式是如何体现这一视角的。我认为考察模式的如下方面将非常有用。

它封装了什么。

它如何使用共性和可变性分析。

它如何将问题域分解为多个责任。

它如何指定对象之间的关系。

它如何展示了从背景设计。

复习题

简答题

1.有些模式具有向什么屏蔽实现的特点？这叫做什么？举出一些例子。

2.举例说明模式有助于思考如何分解责任。

3.学习模式过程中，应该寻找哪5个约束因素和概念？

阐述题

隐藏实现的价值何在？

观点与应用题

1.本书中，已经读到了许多面向对象的重要概念。本章总结了所有这些概念。你觉得获益最大或者最感兴趣的是哪个？

2.Christopher Alexander在书的最后说："在这最后阶段，模式不再重要……，模式已经教会了你对真实的感悟力。"所谓真实，就是模式所描述的关系和约束因素。你感觉自己已经具有了分析手头问题域的新方法了吗？

第26章 参考书目

本章内容

本书是一本入门图书——简介了设计模式、面向对象以及一种更强大的计算机系统设计方法。希望本书能给读者提供一些工具，从此开始用这种更丰富、更有益的方式进行思考。

接下来应该学习什么呢？本书的最后我们将列出一份带有说明的推荐书目。

本章将介绍以下内容。

给出本书配套网站的地址。

为以下几个方面提供推荐书目：

设计模式的进一步阅读；

Java开发人员；

C++开发人员。

希望学习面向对象的COBOL程序员。

一种强大的名为XP（极限编程）的开发方法学的学习。

最后是一些对我个人产生了影响的图书，它们使我相信，生活不仅仅只有编程，更完满的人才能成为更优秀的程序员。

26.1 本书配套网站

网站

学习任何东西都是一个渐进的过程。你我的理解都会随时间而变化。为了能够将自己对软件开发问题（设计模式和其他方面）的最新理解告诉读者，我专门为本书开设了一个网站。网址是：

http://www.netobject-ives.com/dpexplained。

在这个网站，你可以找到本书前面提到的放在网站上的所有信息，以及如下内容。

设计模式的总结，采用一种非常方便的参考格式。

关于敏捷开发、设计模式与极限编程的关系方面的大量信息。

我的公司提供的各种课程的说明，这些课程涉及设计模式、敏捷软件开发、用例（use case）、重构、测试驱动开发和许多其他软件开发相关的主题。

电子杂志

我们还以电子形式每月出版关于软件开发方面各种主题的文章。要订阅这一电子杂志，请访问http://www.netobjectives.com/subscribe.htm。

26.2 推荐阅读

这只是我喜爱的部分图书列表。完整的参考书目请访问本书网站http://www.netobjectives.com/dpexplained。

UML

UML方面，我推荐以下图书。

Fowler M.和Scott K.，UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language，Boston: Addison-Wesley，2000。这是迄今为止我最喜欢的学习UML的资料。不仅可用于入门，而且还是有用的参考书。

面向对象程序设计

面向对象方面我推荐以下图书。

Fowler M.，Refactoring：Improving the Design of Existing Code，Reading，Mass：Addison-Wesley，2000。对重构最全面的论述。

Martin R.，Agile Software Development:Principles，Patterns and

Practices，Upper Saddle River，NJ:Prentice Hall，2002。非常优秀的一本书，既讲述了如何编写面向对象程序，又讲述了如何以敏捷的方式编写。

Meyer B.，Object-Oriented Software Construction，Upper Saddle River，NJ：Prentice Hall，1997。极为透彻的著作，作者是我们行业中最有才华的专家之一。

设计模式

设计模式领域还在发展和深化。可以在不同层次、从不同角度来学习这个领域。我推荐下列图书，希望对你的学习之旅有所帮助。

Alexander C.、Ishikawa S.和Silverstein M.，The Timeless Way of Building，New York:Oxford University Press，1979。无论从个人还是专业而言，本书都是我的最爱。这本书不仅非常有趣，而且充满真知灼见。如果准备只阅读本列表中的一本书，请读这一本。

Alur D.、Malks D.和Crupi J.，Core J2EE Patterns:Best Practices and Design Strategies，Second Edition，Upper Saddle River，NJ:Prentice Hall，2003。本书对J2EE 开发人员和一般意义上从事分布式应用程序的开发人员都非常有用。

Coplein J.，Multi-Paradigm Design for C++，Boston:Addison-Wesley，1998。第2章～第5章必读，即使你不是使用C++的开发人员。正是这本书启发了我们对共性和可变性分析的理解。本书网站上有Jim博士论文的在线版本，此文与他的书是等效的。

Fowler M.，Patterns of Enterprise Architecture，Boston:Addison-Wesley，2002。

Gamma E.、Helm R.、Johnson R.和Vlissides J.，Design Patterns:Elements of Reusable Object-Oriented Software，Boston: Addison-Wesley，1995。现在有些过时了，但是仍然极为有用。

Gardner K.，Cognitive Patterns:Problem-Solving Frameworks for

Object Technology，New York:Cambridge University Press，1998。本书从认知科学和人工智能的角度讨论模式。Gardner博士也深受Alexander著作的影响。

Metsker S.，Design Patterns Java Workbook，Boston:Addison-Wesley，2002。一本很好的模式学习图书。

Nock C.，Data Access Patterns:Database Interactions in Object-Oriented Applications，Boston:Addison-Wesley，2004。数据库领域模式的一本好书。

Schmidt D.、Stal M.、Rohnert H.和Busehmann F.，Pattern-Oriented Software Architecture，Volume 2，New York:John Wiley，2000。本书讨论了多线程和分布式环境相关的一些主题。

26.3 针对Java程序员的推荐读物

学习Java

说到Java的学习，我最喜欢的书如下。

Eckel B.，Thinking in Java，Second Edition，Upper Saddle River，NJ: Prentice Hall，2000。用于学习和参考俱佳。本书配套网站中有这本书可下载版本的链接。

Horstmann C.，Core Java 2 Volume 1 Fundamentals，Sixth Edition，Palo Alto，CA: Pearson Education，2002。另一本学习Java的好书。

Java编程

掌握了Java语言之后，还需要阅读其他一些图书。

Bloch P.，Effective Java Programming Language Guide，Boston:Addison-Wesley，2001。这是一部杰作，它启发我们理解了将使用和构造分离。

Coad P.，Java Design，Upper Saddle River，NJ:Prentice Hall，

2000。如果你是一位 Java 开发人员，这本书是必读书籍。它讨论了在使用设计模式时非常有用的大多数原则和策略，虽然书中并没有特别提及设计模式。

Grand M.，Patterns in Java，Volume 1，Second Edition，New York:John Wiley，2002。如果你是一位Java开发人员，会发现这本书很有用。书中的例子使用Java编写，而且使用了UML叙述。但是，我们相信《设计模式》一书中对约束因素和动机的讨论比本书更有用。当然，有另一组例子还是大有价值的，尤其是它们使用了我们所用的语言（Java）。

Java多线程

Java中处理线程时有一些特殊的考虑因素。我向你推荐如下资源，希望有助于这方面的学习。

Hollub A.，Taming Java Threads,Berkeley，CA:APress，2000。

Hyde P.，Java Thread Programming:The Authoritative Solution，Indianapolis，IN: SAMS，1999。

Lea D.，Concurrent Programming in Java:Design Principles and Patterns，Second Edition，Boston:Addison-Wesley，2000。

26.4 针对C++程序员的推荐读物

C++和UNIX

对于在UNIX上使用C++，我发现了下面这几本必需的书籍。

Eckel B.，Thinking in C++,Volume 1:Introduction to Standard C++，Second Edition，Upper Saddle River，NJ: Prentice Hall，2000。学习C++的最佳图书之一。即使学会了语言之后仍然非常有用。可以从作者网站获得本书的电子版本。

Koenig A.和Moo B.，Accelerated C++:Practical Programming by

Example，Boston:Addison-Wesley，2000。本书提出了一种学习C++的方法：一开始就使用标准库编写完整的程序。是一种非常有益的学习方式。这不仅是一本学习 C++的好书，而且也是一本优秀的教材。遗憾的是，书中对面向对象着墨不多。

Stevens W.，Advanced Programming in the UNIX Environment，Boston:Addison-Wesley，1992。任何 UNIX上C/C++开发者的必备参考书（是的，我知道书中不会讨论面向对象或者模式）[4]。

26.5 针对COBOL程序员的推荐读物

学习面向对象

我发现下面这本书对希望学习面向对象设计的COBOL程序员很有帮助。

Levey R.，Reengineering Cobol with Objects，New York:McGraw-Hill，1995。对于要学习面向对象设计的COBOL程序员，这是一本很有用的书。

26.6 极限编程的推荐读物

学习敏捷和极限编程

要熟练掌握极限编程（XP），我的推荐书目如下。

Beck K.，Extreme Programming Explained:Embrace Change，Boston:Addison-Wesley，2000。本书任何从事软件开发的人都值得一读，即使你并不打算使用 XP。我选择了大约 30 页我认为是必读的章节，列在了NetObjectives网站http://www.netobjectives.com的Resources部分。

Cockburn A.，Agile Software Development.Boston:Addison-Wesley，2001。对于理解敏捷软件开发的各种问题，这是一本非常好的书。

Schwaber K.和Beedle M.，Agile Software Development with Scrum，Upper Saddle River，NJ: Prentice Hall，2001。这是一本读后立即能够派上用场的书。

26.7 程序设计的一般性推荐读物

成为更好的程序员

这本书反映了我的编程观：不断自省而且总是在寻找提高自己和工作的方式。

Hunt A.和Thomas D.，The Pragmatic Programmer:From Journeyman to Master，Boston:Addison-Wesley，2000。这是我每天都要读上几页的趣书之一，其中的许多建议，都可以“有则加勉，无则学之”地采纳。

26.8 个人推荐

超越编程

我相信最好的设计师绝不是那些生活只知道编程的人。相反，能够思考和倾听、具备更完善而深沉的性格、富有思想，才是成为一名优秀设计师所需的。你能够与其他人更好地沟通。你能够从其他学科获得思想（就像我们从建筑学和人类学中获取模式思想那样）。你将创建更加以人为本的系统，毕竟，我们的系统是为人而存在的。

我的许多学生问我喜欢读些什么书，哪些书影响了我的思考方式，哪些书在我的人生旅程中曾经给我以帮助。以下是我的推荐书目。

Alan的书目

Alan推荐以下图书。

Grieve B.，The Blue Day Book:A Lesson in Cheering You Up，Kansas City: Andrews McMeel Publishing，2000。这是一本有趣、令人愉悦的书。情绪低落时，请读这本书。

Hill N.，Think and Grow Rich，New York:Ballantine Books，1960。“富有”不仅意味着金钱——它意味着各种形式的富有。这本书对我个人和事业的成功都有深远影响。

Kundtz D.，Stopping:How to Be Still When You Have to Keep Going，Berkeley，CA: Conari Press，1998。这本书提醒那些工作狂，如何减慢节奏，享受生活，同时仍然完成所有工作。

Mandino O.，The Greatest Salesman in the World，New York:Bantam Press，1968。几年前我阅读并“实践”了这本书。它帮助我按自己一直希望的方式生活。如果要阅读这本书，我强烈建议按照卷轴告诉Hafid的方法去做——不要仅仅阅读它（当你读这本书时，你就知道我的意思了）。

Pilzer P.，Unlimited Wealth:The Theory and Practice of Economic Alchemy，Crown Publishers，1990。这本书介绍了资源和财富一种新范型，以及如何利用它。是一本信息时代的必读图书。

Remen R.，My Grandfather’s Blessings:Stories of Strength,Refuge, and Belonging,New York:Riverhead Books，2000。一本温馨的书，对祈祷的思考。

Jim的书目

Jim推荐以下图书。

Buzan T.和Buzan B.，The Mind Map Book:How to Use Radiant Thinking to Maximize Your Brain’s Untapped Potential，New York:Dutton Books，1994。这本书彻底改变了我教学、与人沟通、思考和记笔记的方式。一种极为强大的技术。我每天都在使用。

Cahill T.，How the Irish Saved Civilization，New York:Doubleday，1995。如果你有爱尔兰血统，这本书将使你无比自豪。野蛮人成了推动文明的最大力量，拯救了欧洲。

Dawson C.，Religion and the Rise of Western Culture，New

York:Doubleday，1950。宗教如何影响了西方文明的发展并阻止了“总是潜伏在表面之下的野蛮主义”。对科学思想非常重要的深刻洞察。

Gerber Michael E.，The E-Myth Revisited:Why Most Small Businesses Don’t Work and What to Do About It，New York:HarperBusiness，1995。如果你正在或者打算自己经营业务，本书将是必读的。它既适合营利企业也适合非营利组织。（Alan附注：作为一家小公司的所有者，我对Jim的推荐深表赞同。）

Jensen B.，Simplicity:The New Competitive Advantage in a World of More,Better,Faster，Cambridge，MA:Perseus Books，2000。思想和知识管理的一次革命。设计更容易使用的系统，在过程和技术中应以人为本。

Lingenfelter S.，Transforming Culture，Grand Rapids，MI:Baker Book House，1998。一种通过“社会博弈论”理解文化的模型。

Spradely J.P.，The Ethnographic Interview，New York:Harcourt Brace Jovanovich College Publishers，1979。任何希望成为优秀访谈者的人的必读之书。所有人类学学生的经典教材。

Wiig K.，Knowledge Management Methods，Dallas:Schema Press，1995。事实上的百科全书，包含了帮助团体更有效地利用其知识资源的所有技术。

---

[1].从背景设计英文有几种称法：design from the context，contextual design和design by context，原书都有出现。中译统一。——译者注

[2].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979，p.545。

[3].Alexander C.，The Timeless Way of Building，New York:Oxford University Press，1979，p.545。

[4].本书有新版。——译者注